

JiVE: JAFMAS integrated Visual Environment

A thesis submitted to the

**Division of Research and Advanced Studies
Of the University of Cincinnati**

In the partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

**in the Department of Electrical and Computer Engineering and Computer Science
of the College of Engineering**

2000

by

Alan K. Galan

B.S.E.E., University of Cincinnati

Committee Chair: Dr. Albert D. Baker

Abstract

JiVE (JAFMAS integrated Visual Environment) is a software tool for the design and implementation of multi-agent systems. JiVE allows a designer to draw a multi-agent system graphically, specify the necessary properties, check the design for correctness, and deploy their application. This can all be done while working collaboratively with designers located elsewhere. JiVE draws on research in agents, multi-agent systems and system design. JiVE offers a set of intuitive, easy to use interfaces that enable a designer to completely specify the agents and agent interactions in a multi-agent system. Also presented in this thesis is an algorithm to check the coherency of the multi-agent system. This algorithm, integrated into JiVE, gives one-touch access to design verification. The compilation of the multi-agent system and the generation of source code, in JAFMAS format, follows design verification. Furthermore, the ability for distributed collaborative work enhances JiVE's features. JiVE pulls from the work in the Computer Supported Collaborative Work research area to provide an application that allows designers whom are not co-located to design a multi-agent system together. JiVE handles all of the issues of concurrency and design consistency, and provides an easy to use collaborative environment. Furthermore, JiVE provides a unique set of features for a multi-agent system design tool. Commonly used techniques in multi-agent system design and collaborative computing, such as web-based collaboration, agent communication languages, and a fully featured Java-based Graphical User Interface are combined in JiVE. This distinct combination of features makes JiVE stand out among the existing multi-agent system design tools.

This thesis presents the research related to the application of the JiVE software, along with descriptions of its design and implementation. A supply chain example exercises the software, conclusions are drawn, and we suggest future improvements and enhancements.

Acknowledgments

I give my thanks and appreciation to Dr. Albert Baker for his assistance and guidance in the preparation of this thesis. I also owe my gratitude for his continuing support in my academic and professional pursuits.

My thanks go to my thesis committee, Dr. John Franco, Dr. Ali Minai and Dr. Marios Polycarpou, who have made the completion of this thesis possible.

I would also like to thank my parents for their undying love and support from beginning to end. They have contributed much to my life in many ways.

To my friends and family, I extend my thanks to each of you; your various contributions to my life have meant a lot.

Table of Contents

Table of Contents	<i>i</i>
List of Figures	<i>iv</i>
List of Tables	<i>v</i>
1. Introduction	1
1.1 Background	1
1.1.1 Agents	1
1.1.2 Multi-agent Systems	1
1.1.3 Multi-agent System Design Tools	2
1.1.4 JAFMAS	2
1.2 Objective	2
1.3 Overview	3
2. Literature Review	4
2.1 Agents and Multi-agent Systems	4
2.1.1 Specification of Conversations and Conversation rules	4
2.1.2 Agent Communication Languages	5
2.2 JAFMAS	6
2.3 Multi-agent System Design Tools	7
2.3.1 AgentBuilder	7
2.3.2 ADE	7
2.3.3 ZEUS	8
2.3.4 DECAF	8
2.3.5 MASSYVE Kit	8
2.3.6 JADE	8
2.3.7 KAoS	9
2.3.8 Bond	9
2.3.9 Comparison of Existing Design Tools with JiVE	9
2.4 Collaboration	11
2.4.1 Requirements	11
2.4.2 Architecture	12

2.4.3	Concurrency	13
2.4.4	Example Groupware Systems	13
2.5	Summary	15
3.	<i>System Design</i>	17
3.1	System Requirements	17
3.1.1	Collaborative Support	17
3.1.2	Graphical Interface	17
3.1.3	Design Validation	18
3.1.4	Source Code Generation	18
3.2	System Design	19
3.2.1	Access Control	19
3.2.2	Graphical Representations	20
3.2.3	Petri-Net Analysis	26
3.2.4	Code Generation	28
3.3	System Implementation	29
3.3.1	Architecture	29
3.3.2	JiVE Server	30
3.3.3	JiVE Client	31
3.3.4	JiVE Communication	33
3.3.5	Agent Entity Representations	36
3.4	Summary	38
4.	<i>Example</i>	39
4.1	Supply Chain	39
4.2	Project Initialization	39
4.3	Designing the Agents	40
4.4	Designing the Conversations	42
4.5	Designing the Conversation Rules	44
4.6	Analyzing Conversations and Generating Code	47
4.7	Collaboration	48
4.8	Using the Design	49

4.9	Summary	50
5.	<i>Discussion and Future Work</i>	51
5.1	Summary	51
5.2	Discussion	51
5.3	Future Work and Enhancements	52
5.3.1	Improved Collaboration features	52
5.3.2	Better Graphical Interface	53
5.3.3	Improved Efficiency During Design	53
5.3.4	Better Project Support	54
5.3.5	Improved Analysis	55
5.3.6	Improved ACL Support	55
5.3.7	Improvements in Agent Design	55
5.4	Summary	56
	<i>Bibliography</i>	58
	<i>Appendix A - Users Manual</i>	A1
	<i>Appendix B - JiVE Source Code</i>	B1
	<i>Appendix C - Supply Chain Example Source Code</i>	C1

List of Figures

Figure 3-1 - JiVE Agent Representation	20
Figure 3-2 - JiVE Agent Conversation Representation	21
Figure 3-3 – Multi-Agent System Design Window.....	23
Figure 3-4 - Conversation Design Window.....	24
Figure 3-5 - General JiVE Architecture	29
Figure 3-6 - Server Architecture	30
Figure 3-7 - Client Architecture.....	31
Figure 3-8 - Object Relationship.....	37
Figure 4-1 - Login Window	39
Figure 4-2 - Designing Agents.....	40
Figure 4-3 - Adding a Property	41
Figure 4-4 - Agent Properties.....	41
Figure 4-5 - Customer Agent's Conversation Design	42
Figure 4-6 - Logistic Agent's Conversation Design	44
Figure 4-7 - LogExecNet Conversation Rule Design	46
Figure 4-8 – Specification of Message Sender/Recipient.	46
Figure 4-9 - Such That Property Design	47
Figure 4-10 - Analysis Results.....	48
Figure 4-11 - Edit Control Request Dialog Box.....	49

List of Tables

Table 2-1 - Comparison of JiVE with Existing Multi-System Design Tools	10
Table 3-1 - List of JiVE Menu Items	25
Table 3-2 - List of Dialog Boxes and Their Functions	32
Table 3-3 - Message Types	35
Table A-1 - List of JiVE Classes.....	A2

1. Introduction

JiVE (JAFMAS integrated Visual Environment) allows a designer to draw a multi-agent system graphically, specify the necessary properties, check the design for correctness, and deploy their application. This can all be done while working collaboratively with other designers located elsewhere.

1.1 Background

1.1.1 Agents

The definition of an agent is a topic that provides for much discussion. In general, an agent is a piece of software, possibly with some intelligence, with a well-defined role and a set of goals; it can even be autonomous or pro-active. Regardless of what properties the agent has, it will work on the behalf of a user providing some service or data. The concept of software working on our behalf is very appealing. More importantly is that the agent can work more efficiently than a person and can take on tasks that would ordinarily not be appealing. These tasks can include things that are highly repetitive, computationally intensive, or just plain mundane.

1.1.2 Multi-agent Systems

A multi-agent system is a conglomeration of agents, which when acting together exhibit a greater system level behavior. The designer programs this behavior in some cases and in other cases this can be an emergent behavior. The system level behavior benefits from the individual actions of the agents.

Numerous multi-agent systems have been deployed both in academia and industry, ranging from patient scheduling in a hospital [Aknine and Aknine 1999] to climate control of a building [Ygge and Akkermans 1997], in areas as varied as Information Broadcasting via the Internet [David and Kraus 2000] and supply chain integration [Nissen and Mehra 1999].

The benefits of a multi-agent system are many and in most cases can include flexibility, scalability, decentralization, and robustness.

1.1.3 Multi-agent System Design Tools

The complex nature of agents, and even more so, multi-agent systems, suggests that additional tools are required to assist in the design and implementation process. This is true for all but the simplest of agents and systems. The tools that we call on to assist us range from the low level frameworks to the higher-level integrated development environments. Each level of abstraction allows for a unique view into the construction and deployment of multi-agent systems. They each also provide a unique set of benefits to the system. The lower level frameworks provide a designer assistance for a certain subset of design issues, for example handling communication or agent mobility. The higher-level tools provide an abstracted view of the multi-agent system, allowing the designer to concentrate on the macro level behavior, leaving the lower level details to the design tool. As is the case with JiVE, the higher-level design tool can leverage a lower level tool to provide a comprehensive solution.

1.1.4 JAFMAS

The Java-based Framework for Multi-Agent Systems (JAFMAS) [Chauhan 1997] provides the basis for JiVE and resulting multi-agent system design. JAFMAS is a methodology and architecture for the design and implementation of multi-agent systems in Java. It focuses on communication, agent interaction, and multi-agent system coherency and coordination.

JiVE enhances the use of JAFMAS by providing a visual environment for the design of the agents and their communications. It removes the difficult and time-consuming task of manually writing code to support communication between agents in a multi-agent system. Furthermore, JiVE guides the designer through the JAFMAS methodology and automates the analysis of the conversation model before generating source code in JAFMAS format.

1.2 Objective

JiVE meets the following objectives:

- Collaborative support for multiple, distributed users.
- A graphical interface for the design and visualization of multi-agent systems.
- Automated design validation and error checking for multi-agent system designs.

- Generation of usable software code for the deployment of multi-agent systems.

In addition to those main objectives, other aspects are considered. The JiVE architecture is a client/server application allowing multiple users the ability to login to a design session. The use of the Java™ programming language and the Java Foundation Classes in the development of JiVE provides platform independent user interfaces.

1.3 Overview

Chapter 2 introduces multi-agent systems, multi-agent based design tools, and collaborative programming environments and shows the relationship between JiVE and this previous work. Chapter 3 then elaborates on the design and architecture of JiVE. Chapter 4 presents an example design, employing the JiVE software, and exercising its major features. Chapter 5 provides a statement of future work and improvements for the software, along with conclusions drawn from this work. Included as appendices are: the JiVE Users Manual (Appendix A), the source code for JiVE (Appendix B) and source code generated from the example design (Appendix C).

2. Literature Review

This chapter first presents a discussion of intelligent agents and multi-agent systems. It then provides a survey of tools for the design and implementation of multi-agent systems. The chapter concludes with a discussion of the relevant issues related to collaboration.

2.1 Agents and Multi-agent Systems

The definition of an agent is a popular debate in the agent community. There are however various terms that seem to consistently come up when defining an agent. These can include intelligence, autonomy, reactivity, and perception. Huhns and Singh give a definition for an agent that reflects this ongoing debate:

“Agents are active, persistent (software) components that perceive, reason, act and communicate.

Some researchers add further properties, such as being autonomous, goal directed, reactive, or declaratively programmed. Others limit agents to the role of representing a user or database.” [Huhns and Singh 1998]

Given these disparities, it remains that an agent should provide a service to the user, working toward some goal while representing the user’s interests.

Linking these possibly intelligent and/or autonomous agents together forms a conglomeration of agents termed a multi-agent system. This type of system is characterized by multiple independent entities working towards a common goal, or solving a problem or set of problems. They interact within themselves and possibly with the environment and/or outside entities.

2.1.1 Specification of Conversations and Conversation rules

The cornerstone of a multi-agent system is the communications. Conversations allow the agents to interact and dictate the behavior of the system. The specification of the conversation rules is an important aspect of the design and realization of multi-agent systems. A graphical representation usually provides the easiest description of agent conversations. The tendency is to resort to a formal methods approach to specifying the conversations. The most common means of representation are Petri-nets or state machines.

COOL [Barbuceanu and Fox 1995] presents an example of the specification of communications, where they use a finite state machine to represent each agent's communication parameters. Chauhan further enhances this model [Chauhan 1997] to include a Petri net representations taken from the individual conversations, aggregating them into one specification.

Moldt and Wienberg present a representation of multi-agent systems based on Colored Petri nets [Moldt and Wienberg 1997]. The tokens in the CPN represent the dynamic components of the complex system. They further make the distinction between the agent society and the agent, both of which use Petri nets as representations. Cost et al. have also used Colored Petri nets [Cost et al. 1999]. They claim that the use of these CPNs allow for improved support of concurrency and complex interactions.

In contrast, Parunak presents an enhanced form of the Dooley Graph [Parunak 1996]. The Enhanced Dooley Graphs are said to be useful in capturing agent interactions and expressing them in an easy to understand manner. The Z specification language has also been proposed as providing a framework for describing multi-agent interaction [d'Inverno, Kinny and Luck 1998].

2.1.2 Agent Communication Languages

The agent communication languages (ACL) used in multi-agent systems are another important aspect of the design of a multi-agent system. An ACL is a precisely defined protocol for inter-agent communication in multi-agent systems, including syntactic and semantic definitions. The two ACLs that seem to be in the forefront of acceptance as standards are KQML and FIPA-ACL.

KQML is a message format as well as a message handling protocol. KQML consists of three layers; the content layer, the message layer, and the communication layer. The content layer consists of the content of the message. The communication layer handles the lower level communication parameters. The message layer consists of a performative, which conveys information about the content of the message, as well as optional information to describe the content language [Labrou and Finin 1998].

FIPA-ACL is another ACL with a format similar to that of KQML. ACL messages use a set of parameters, the first representing the communicative act of the message. The additional parameters specify items: to aid in message delivery, to assist in the interpretation of the content, and to allow the receiver to respond co-operatively [FIPA 1997].

2.2 JAFMAS

As the title of the software indicates, JiVE builds heavily on JAFMAS (Java-Based Framework for Multi-Agent Systems) [Chauhan 1997]. JAFMAS provides a methodology and a framework for designing multi-agent system. JiVE uses the five-step methodology provided by JAFMAS:

- Identify the Agents
- Identify the Conversations
- Identify the Conversation Rules
- Analyze the Conversation Model
- Implementation of Multi-agent system

The JAFMAS software architecture provides a set of classes for the implementation of the multi-agent system, as proscribed by the methodology. These classes provide support for the sending and processing of broadcast and directed communication, conversation execution, and system coordination. JiVE integrates with JAFMAS by allowing the designer to step through the methodology using a set of graphical user interfaces, with the result being a set of generated class files that are compiled into a multi-agent system.

JAFMAS has a unique construct, carried forward into JiVE that warrants additional description. Each agent in a JAFMAS-based multi-agent system can have a list of ‘Required Resources.’ These ‘Required Resources’ establish the needs of the agents in the system, and allow for the discovery of agents that provide that resource. When the multi-agent system is started the agents use multi-cast messages to find the other agents in the system that provide their ‘Required Resources’.

2.3 Multi-agent System Design Tools

There are various design tools available for the design of multi-agent systems. The development of these design tools range from commercially available products to research prototypes. We focus first on tools that provide graphical interfaces for the development of the multi-agent systems, and then we mention several tools that do not provide a graphical design environment, but are of particular relevance to this work.

2.3.1 AgentBuilder

AgentBuilder is a commercial product produced by Reticular Systems, Inc. it provides an integrated toolkit for the construction of intelligent agents. AgentBuilder provides a fully featured set of graphical interfaces for the design and development of a multi-agent system. It provides various graphical tools including project management, ontology management, agent management, protocol management, and run time engines. AgentBuilder is Java-based and its communication language is KQML.

The AgentBuilder agent model consists of agents that have beliefs, capabilities, commitments, intentions, and behavioral rules. Given an agent model's initial state and external influences the agent operates and communicates with the external world, which in turn modifies its states [Reticular 1999a]. An interesting agent based system designed using AgentBuilder is a prototype system for buying and selling power generation capacity [Reticular 1999b].

2.3.2 ADE

Another commercial multi-agent design tool is ADE (Agent Development Environment) by Gensym. ADE is an Integrated Development Environment for Distributed Multi-Agent Applications, which is built on G2, an Object-Oriented graphical environment. ADE provides a graphical language called AdeGrafcet (an extension of the French Grafcet standard, which is based on Petri net representations) for the specification of the agent's behavior. ADE provides a communication middleware, utilizing subject-based addressing. ADE also has a simulation environment for simulating the design before implementation, as well as a debugging environment [Mehra and Nissen 1998].

2.3.3 ZEUS

ZEUS is a toolkit for building distributed multi-agent systems. ZEUS specifies a design approach utilizing a three-layer model of an agent, the definition layer, the organization layer, and the co-ordination layer. It provides a set of graphical interfaces to allow a designer to create the multi-agent system. An agent is comprised of libraries that include communication (using KQML), social interaction, multi-agent coordination, and planning and scheduling. This tool generates source code for the agent-based system in Java [Nwana et al, 1999].

2.3.4 DECAF

DECAF (Distributed, Environment-Centered Agent Framework) is a software toolkit for developing intelligent agents. It provides user interfaces for the specification of the agent behavior, including logical and temporal constructs. It allows reuse of generic agent behaviors. Its Plan-Editor removes the designer of a multi-agent system from the low level API and allows for the rapid development of agents, which are generated in Java. DECAF also provides some design time error checking, and supports the use of agent communication languages [Graham and Decker, 1999].

2.3.5 MASSYVE Kit

The MASSYVE (Multiagent Agile Manufacturing Scheduling Systems for Virtual Enterprise) kit is a software kit for the development of multi-agent systems. It allows a user to configure, modify, and create a multi-agent system; it provides user interfaces to guide a designer through the process. It also provides mechanisms to launch the agents, both locally and remotely [Jacomino, Geszychter and Rabelo 2000].

2.3.6 JADE

JADE (Java Agent DEvelopment Framework) is a software framework for developing FIPA-compliant multi-agent systems. JADE provides a set of interfaces for the design of agents, implemented in Java. JADE uses the FIPA-ACL, utilizing a combination of socket, RMI and CORBA. This framework also provides a set of pre-defined behaviors for the use of the

designer, including ‘SimpleBehaviour’, ‘CyclicalBehaviour’, and ‘NonDeterministicBehaviour’ [Bellifemine, Poggi, and Rimmassa 1999].

2.3.7 KAoS

KAoS is an agent architecture that provides a form of agent-oriented programming. A set of structures: knowledge, desires, intentions, and capabilities represent the agents in KAoS. State transition diagrams specify the conversation policies [Bradshaw et al. 1997].

2.3.8 Bond

Bond is a Java-based, object-oriented middleware for collaborative network agents. It presents a set of objects to enable agent communication, including Message Passing, Object Replication, and Directory Services. The bond agents communicate using KQML [Bölöni 1998]. A “multi-plane state machine” represents the behavior of the bond agents [Bölöni and Marinescu 1999].

2.3.9 Comparison of Existing Design Tools with JiVE

These design tools use various approaches to facilitate the design of multi-agent systems. Among these, are some aspects that are notably similar to JiVE and other areas where JiVE improves on the current collection of work.

JiVE, like many of these other design tools, has turned to Java to provide platform independence. This choice affects the deployment of the design tools themselves as well as the systems that these tools design. JiVE, is both written in Java and produces Java based multi-agent systems.

Many of the design tools listed support the standard ACLs, either KQML (AgentBuilder, Zeus, and Bond) or FIPA ACL (JADE). JiVE, which supports the use of both ACLs, combines ACL support with the ability to have user defined communication languages.

JiVE uses a similar representation for the agent communications as many of the other tools. Various tools use state machines, Petri-nets, or variations thereof to specify the agent communication. The “multi-plane state machine” used in Bond is an interesting expansion of the traditional state machine representation.

Checking the conversation models for consistency and coherency is not a feature that many design tools have, although DECAF does have some design time error checking and ADE provides some simulation and debugging capabilities. JiVE on the other hand has an algorithm for converting the agent communication representations to a Petri-net for analysis of consistency and coherency of the systems communications.

The generation of source code for the deployment of the multi-agent systems is another important aspect of JiVE, which is included in some of the current design tools, including Agent Builder ZEUS and DECAF. Others directly deploy the agent based system, including ADE, MASSYVE Kit, KAoS and Bond. The generation of source code allows the users to add behavior to the agents after using the tool for the basic design, where a direct deployment prohibits this. However, using direct agent deployment the design tool can better control the final system, albeit at the expense of flexibility.

JiVE is the only design tool available to support multi-agent system design and development in JAFMAS. All of the design tools are very tightly integrated with their own underlying multi-agent framework; JiVE however has the ability to incorporate other frameworks than JAFMAS by replacing the Code Generation module. Additionally, JiVE is the only tool that supports multi-user development and a web-based interface.

	Collaboration Support	Graphical Design Environment	Design Validation	Implementation Support†	ACL Support‡	Java-Based	Pricing*
JiVE	✓	✓	✓	C	F, K, G	✓	F
AgentBuilder		✓	✓	C	K	✓	\$800-\$5,000
ADE		✓	✓	D			N/A**
ZEUS		✓		C	K	✓	F
DECAF		✓	✓	C		✓	F
MASSYVE		✓		D			F
JADE		✓		D	F	✓	F
Kaos			✓	D		✓	F
Bond				D	K	✓	F

Table 2-1 - Comparison of JiVE with Existing Multi-System Design Tools

† C = Code generation, D = Direct deployment

‡ F = FIPA-ACL, K = KQML, G = General speech-act based messaging.

* F = Freeware: Free for most non-commercial uses.

**** ADE is a commercial development environment sold bundled with services.**

2.4 Collaboration

Collaboration of multiple users is an interesting topic with a significant amount of research behind it. This research generally falls under the title Computer Supported Cooperative Work (CSCW). The implementation of CSCW concepts is usually termed groupware. Groupware applications improve the ability of a group to produce results by allowing them to collaborate and more importantly to be an active participant in a distributed design environment. Mills recognizes a future for groupware applications, specifically in Collaborative Engineering, due to the high return on investment and improving Internet standards [Mills 1998].

This section discusses the requirements for CSCW tools or groupware. There is also a discussion of issues involving the architectures needed. A section on the concurrency of data in these tools follows that. We then provide several examples of specific groupware applications and conclude with specific comparisons to JiVE.

2.4.1 Requirements

Some methods by which requirements for groupware applications are established are outlined in [Ehrlich 1999] and summarized in this subsection.

2.4.1.1 Work Practices

Work practices refer to the way in which a user of an application utilizes that tool in real life situations. The development of a collaborative work environment, or any application for that matter, should of course consider this. A realization is needed that designing an application should keep the needs of the user in mind. Furthermore, [Hughes et al. 1996] suggests that the practices of a user can adapt over time in response to a technology. In many aspects, this is the opposite of what an application developer should desire. An application should leverage the natural way that a user thinks about the work involved and design towards that end, enhancing the user experience, not hindering it.

2.4.1.2 Methodologies

Some methodologies for establishing the work practices involved in a groupware application are ethnography, participatory design, and action research [Ehrlich 1999].

Ethnography refers to the act of observing the end users of an application in action to evaluate their work practices. This can involve watching their actions, interviewing and participation. The application of these findings can have very positive impacts on the application design. Participatory design allows the end users to be actively involved in the design process. Action research also involves observing the end users of an application, but in an attempt to improve the processes through the implementation of the technology.

2.4.2 Architecture

Greenberg and Roseman discuss the merits of centralized vs. replicated architectures [Greenberg and Roseman 1999]. A centralized architecture is one that has a main application located on a central server machine. Clients submit request to the server and display the results based on the server's output. In this case, the server can handle all concurrency and synchronization locally. All client requests are serialized such that there will be no inconsistencies in the data. The downfalls of this type of centralized system include the server being a computational bottleneck, and network latency.

Replicated architectures have no centralized server and respond to changes in state locally and send information to each other in order to update the other's data. With a distributed system, the cost of a replicated architecture is a more complicated system. There is also now a need for distributed concurrency control. Some benefits of this approach are a decrease of the computational bottleneck due to the server, and less chance for latency due to the diminished amount of data transferred.

The compromise, in many cases, is a hybrid architecture incorporating centralized and replicated features in an attempt to gain the best of both worlds. Patterson, Day, and Kucan discuss an example of this type of architecture [Patterson, Day and Kucan 1996]. They talk of an architecture that has a central server that acts as a repository for the shared system data and handles any changes for the clients. The clients however control the data locally and manage

their own view of the data. JiVE uses this hybrid approach to its architecture, but leans towards a centralized approach (c.f. Section 3.3).

2.4.3 Concurrency

An important aspect of designing a groupware application is the concurrency of the data in the application. Prakash suggests two broad classes of concurrency control in a group editor [Prakash 1999].

Pessimistic concurrency control refers to the method of acquiring network wide locks before allowing the editing of a document or section of the document, which incidentally is the method used by JiVE. This method can slow the rendering of the local data in cases where the lock has to be acquired. Optimistic Concurrency Control allows any changes on the local copy of the data to be effective immediately and then sends that information over the network (including a time stamp) to be reconciled with all of the other users. When this information is reconciled with the operations in each client, there may be a need to roll back or undo operations. This method requires that operations be reversible, in order for the reconciliation at each client. In addition, there is the opportunity for the users to interact with inconsistent data.

2.4.4 Example Groupware Systems

2.4.4.1 JPECS

JPECS (Java Programming Environment with Collaborative Support) is a web-based, platform-independent collaborative work environment. It provides a platform for the distributed editing and simulation of a visual programming language (in this case a finite state machine simulator), including a whiteboard and chat feature. It deals with issues such as concurrency, multi-user communication, and collaboration [Inamdar 1997]. JPECS provides a basis for JiVE's collaborative features. JiVE utilizes a similar approach in terms of architecture and messaging to that of JPECS. JiVE does improve upon access control issues of a collaborative programming environment. JiVE provides a more fine-grained view of access control entities in the system and makes the access control more transparent (c.f. Section 3.2.1). JiVE does not though

provide a full set of user communication tools, such as a whiteboard or a chat-board and does no user privilege control.

2.4.4.2 JSIM

JSIM (Java Simulator) was built on JPECS [Inamdar 1997] and provides a collaborative tool for the definition and simulation of control systems. JSIM provides a multi-user communication utility and a block diagram editor. There is also a conference management utility to allow multiple design sessions to run concurrently [Chen 1998]. JSIM, like JPECS, provides a basis for the JiVE architecture and user control facilities.

2.4.4.3 Joint EMACS

Joint EMACS is an EMACS-style text editor that allows for the collaborative editing of documents. It supports a high degree of fine-grained mutual awareness, and has well-developed methodologies for handling concurrency control [Ressel, Nitssche-Ruhland and Gunzenhäuser 1996]. In addition to the comprehensive handling of concurrency control, which JiVE also uses, Joint EMACS provides mechanisms for ‘Undo’ operations, which could be a useful feature in a tool such as JiVE.

2.4.4.4 TeamRooms

TeamRooms is a collaboration tool that allows for collaboration either in real time or asynchronously. This implies a persistence of data in the environment. As part of the environment they provide various collaboration tools, including, user and conference management and visibility, graphical editors, textual editors, and chat boards [Roseman and Greenberg 1996]. JiVE uses a similar client/server architecture, including user verification and a data repository, though it does not incorporate any awareness features or collaboration tools beyond agent editing. TeamRooms also, adds an Administration client, which would be an interesting expansion of the JiVE functionality (c.f. Section 5.2).

2.4.4.5 Artefact

Artefact is a web-based tool for building collaborative applications that is itself a distributed collaborative tool. It uses a client/server architecture, which uses HTTP as its communication protocol. There are a number of base 'objects' that a user can use to design their application as well as the ability to develop customized application objects [Brandenburg et al. 1998]. Artefact addresses some important collaboration issues which are applicable to JiVE, such as speed and performance of web-based collaboration, including Java, but relies on a more centralized approach and a lightweight Java applet with HTTP enhancements. In addition, Artefact allows the user to interact synchronously or asynchronously with the application.

2.4.4.6 Summary of JiVE's Relationship to Existing Groupware Systems

JiVE's collaboration features draw from existing work on groupware systems. JPECS and JSIM provide much of the basis for JiVE's collaborative architecture, messaging and user control. Concurrency control components of Joint EMACS are similar to JiVE, although more fully developed. In terms of architecture, JiVE, TeamRooms and Artefact are similar in their client/server configuration. In addition, Artefact provides a thorough approach to web-based collaboration from which future versions of JiVE could draw. All of the tools mentioned in this section have various collaboration tools that could supplement JiVE's collaboration experience.

2.5 Summary

This chapter discussed some of the current literature in the fields related to this thesis. The chapter started with some background on agents and multi-agent systems. Specifically addressed were the communications between agents, including the basis for JiVE's representation of conversation and conversation rules and the use, by JiVE of ACLs. The presentation of the existing work in the area of multi-agent design tools highlighted the unique approach JiVE takes on this issue. The distributed collaboration features of JiVE are one of the things that make it stand out. The existing work in collaborative software provides the bases for these features. This includes the requirements for a groupware application and some design methodologies. We also showed how JiVE relates to existing work in terms of architecture and

concurrency control. Finally, we provided a comparison of JiVE and existing collaborative applications. The next chapter presents the design and implementation of JiVE.

3. System Design

This chapter discusses JiVE's design and implementation. Highlighted are the important aspects for such a design tool, followed by a discussion addressing these aspects. We then look at the implementation details of the JiVE software, starting with a higher level view of the architecture, and continuing to some of the lower level details.

3.1 System Requirements

The desired multi-agent system design tool has the following elements.

- Collaborative support for multiple users, in a web-based environment.
- A graphical interface for the design and visualization of the multi-agent system.
- Automated design validation.
- Source Code generation and regeneration

3.1.1 Collaborative Support

Having multiple users work on the same project at the same time in different locations is a very appealing notion. In order to realize the goal of having multiple users, there must be a mechanism to prevent two users from changing the same thing at the same time, which could lead to data inconsistencies. This editing control should prevent any collisions or inconsistencies, while still allowing the users to collaborate effectively, and be unobtrusive.

We not only want to collaborate on a design, we would like to be able to do so in a distributed manner. This type of distributed collaboration will have an impact on the architecture used for the software tool. In addition, we will need to provide facilities for communication connections.

3.1.2 Graphical Interface

The first, and arguably the most important, aspect of designing a multi-agent system design tool is the graphical representations used to denote the different entities in the system, as well as the layout and design of the graphical user interfaces.

The graphical interface that the development environment employs should make the design of a multi-agent system intuitive and provide a premium of information to the designers without making it too difficult to understand and/or find the information. The information conveyed by the graphical representations of the agents and their conversations should allow the user to see at a glance the basic interactions and properties in the system. When desired by the user, there should be a mechanism for getting more information about any one piece of the design.

The layout of the graphical user interfaces should also be intuitive to the user. Buttons and messages should be easy to understand and help access should also be easy. Additionally, the user should be able to see the status of the design process at any given time.

3.1.3 Design Validation

The conversations of a multi-agent system dictate the behavior of the system. In order to ensure that the multi-agent system will exhibit the desired system level behavior, one must ensure that the interactions between agents are consistent with the goals of the system. This means that the conversations that an agent can engage in must be logically consistent and coherent. Therefore, it is necessary to specify the conversations in such a way that we can prove for the entire system the consistency and coherency. So, our multi-agent system design tool shall include conversation analysis techniques. The results from this analysis should help guide the user on how to fix any logical errors in their design. This type of validation will help the designer in creating a multi-agent system that achieves its desired goals [Galan, Baker 1999].

3.1.4 Source Code Generation

Once the multiple, distributed users have created a multi-agent system, and have checked the design for logical consistency, the next step is to assist them in the implementation. The design tool should be able to take the specified agents, conversations, and properties and translate them directly into usable source code. This source code may need manual supplementation, but the designers should have a solid starting point for enhancing their design. Additionally there should be support for regeneration of the source code, to accommodate design changes, even after the designer has made manual modifications.

3.2 System Design

This section outlines how JiVE meets these requirements.

3.2.1 Access Control

An important aspect in designing a collaborative environment is to control the changes that various users can make. JiVE attempts to control this with as little intrusion as possible. In many cases, the system handles access control automatically. In other cases, the user has to manually request control from another user, who can relinquish that control at their leisure. JiVE has built in mechanisms for this request/relinquish scheme.

One way to accommodate multiple users working on the same project collaboratively would be to limit editing access of the whole project to one user at any given time. This approach does not provide the flexibility desired for JiVE. Instead, we use a more fine-grained access control model. This means that access control is applicable at several different levels, namely:

- Agent Edit Control
- Conversation Edit Control
- Conversation Rule Edit Control

That means that any one user can have a lock on any one of the entities listed, while other users can have the editing control for other entities. For example, ‘User1’ could have control of ‘Agent1’, while ‘User2’ could have control of ‘Conversation1’, which may be a conversation of ‘Agent1’. Of course, when any user in the system makes a change to any part of the design, this change is visible to every other user in the system.

The automated access control follows the following rules:

- As a user adds any controlled entity to the system, they automatically get access control.
- If no other user has access control of an entity then any user who attempts an editing operation automatically receives access control to that entity.
- If a user has control of an entity then other users are prohibited from making changes. In this case, the users can use JiVE to request control from the user who has it. That user can relinquish control or ignore the request.

- When a user logs out of the system and/or the client loses the connection with the server, that user relinquishes Edit Control of all entities.

The source code that handles access control for the entire project is contained in the *Repository* class (which manages all project data), See Appendix B. As mentioned above, there are several levels of access control, and the model used for each is analogous. For illustrative purposes the methods for agent edit control are presented here:

- `checkEditControlOfAgent(AgentRepresentation agent, User user)` - This method is called from any place in the code where a change might be made to an agent. If no one has edit control of the agent, they are automatically granted the control. If the user that is attempting to make the change has edit control, then the change is allowed to go forward. Finally, if the user attempting to make the change does not have edit control, then the change is cancelled.
- `requestEditControlOfAgent(String agentName, User userWhoWantsControl)` - This method is called in response to a user asking for edit control of an agent. If no other user has the edit control than the control is granted to the requester, otherwise a message is sent to the user who does have control, asking him to relinquish it.
- `relinquishEditControlOfAgent(String agentName, User userWhoHadControl)` - This method is called in response to a user relinquishing edit control of the agent.

3.2.2 Graphical Representations

In order to draw a multi-agent system in a graphical manner, we need to intuitively represent the agents and their conversations. In JiVE, a rectangle on the Agent editing screen represents the agent. This rectangle lists the name of the agent, and an ‘expand’ icon, along with the number of intended subscription/publication channels. When expanded one can see the relationship between the agent and the various subjects with which it communicates. For example, in Figure 3-1 we see a depiction of an

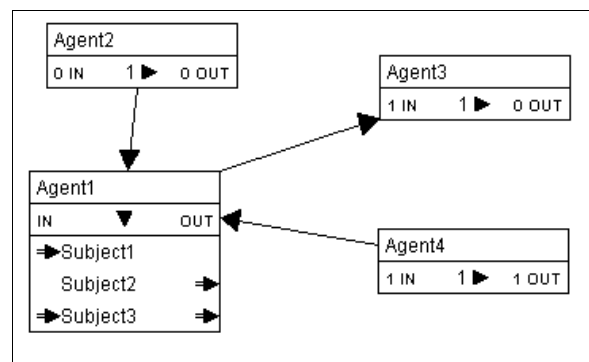


Figure 3-1 - JiVE Agent Representation

agent. This agent's name is 'Agent1'. The subjects that it subscribes to and/or publishes to are listed below it on the expanded listing. We can see that 'Agent1' receives messages on the 'Subject1' subscription channel, sends messages on the 'Subject2' publication channel and both sends to and receives from the 'Subject3' subscription/publication channel. Also noted in this graphic are the directed communications that this agent has with other agents in the system. We see two arrows going out (to 'Agent3' and 'Agent4') and two coming in to the agent (from 'Agent2' and 'Agent4'). With this information for each agent appearing on the agent editing screen, we can see the interactions between the agents, but are not overwhelmed by details about the substance of those communications. To view and edit an agent's properties (c.f. Section 4.3) a dialog box can be opened. Similarly, to find out about or edit an agent's conversations, a conversation editor window can be opened.

Each conversation for each agent in the multi-agent system is represented by a finite state machine, enclosed in a box, and labeled with the conversation's name. Each state in the finite state machine represents

one possible state of the conversation and each transition in the finite state machine represents a means for changing states in a conversation, called a conversation rule. A dot in a state denotes the starting state

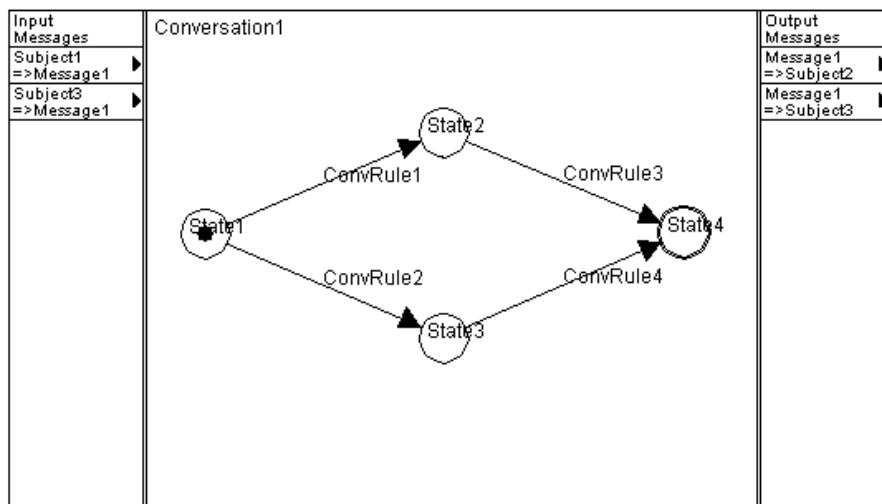


Figure 3-2 - JiVE Agent Conversation Representation

state drawn with a doubled line indicates a final state for the conversation.

Figure 3-2, for example, shows an agent's conversation named 'Conversation1'. This conversation has four different states. Of those four states, 'State1' is an initial state, 'State4' is a final state, and 'State2' and 'State3' are neither an initial nor a final state. This conversation

has four conversation rules, 'ConvRule1', 'ConvRule2', etc. 'ConvRule1' for example points from 'State1' to 'State2', this denotes that if the conversation is in 'State1' and the conditions for 'ConvRule1' are met, then the messages and/or computations associated with 'ConvRule1' are executed and 'Conversation1' would transition to 'State2'. If the user desired more information on any conversation rule, they can expand that rule and view all of its properties.

Figure 3-3 shows JiVE's main graphical user interface (GUI) and Figure 3-4 shows the Conversation Design Window. The use of each of these design windows is described as a part of the example given in Chapter 4. The features of these windows are described here.

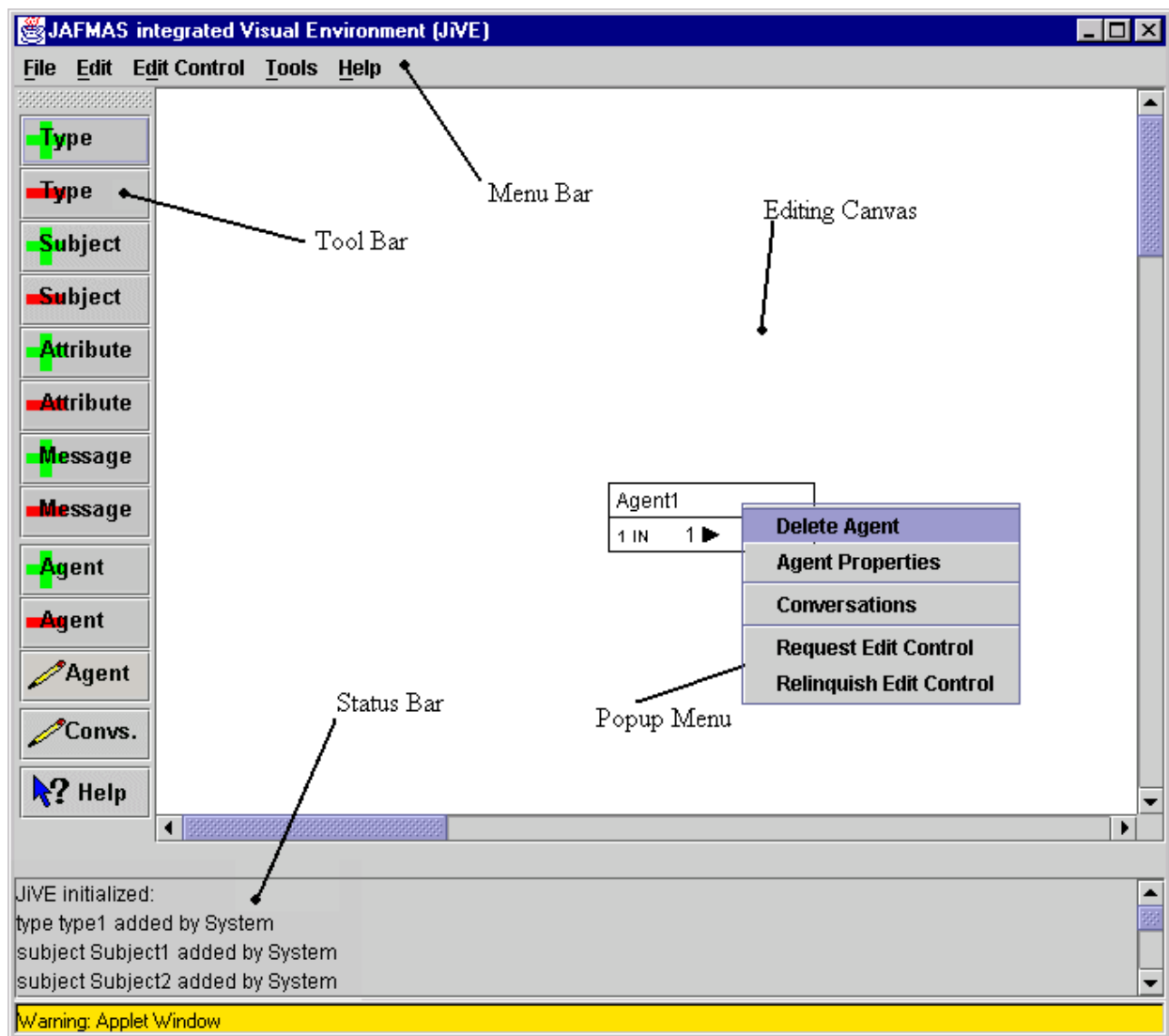


Figure 3-3 – Multi-Agent System Design Window

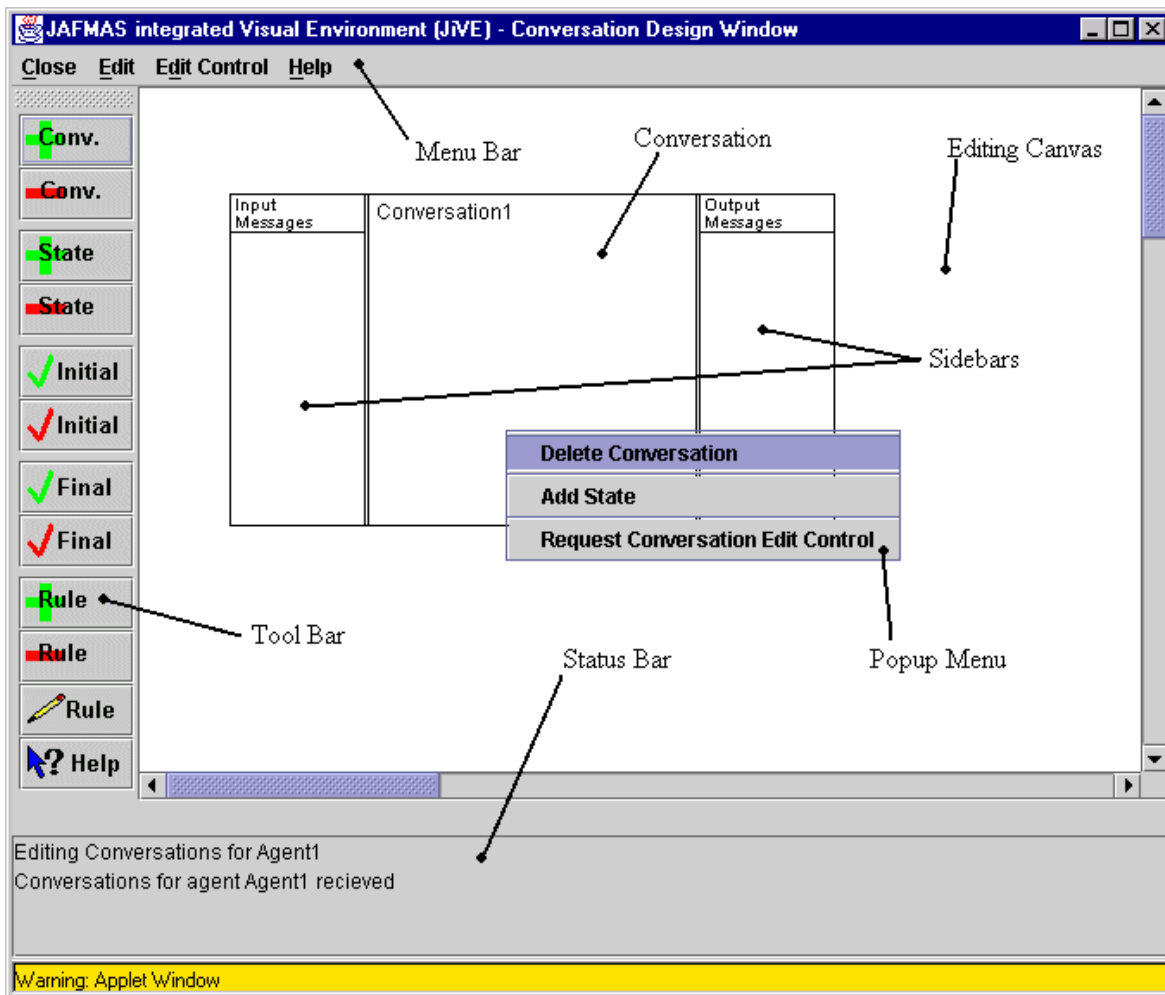


Figure 3-4 - Conversation Design Window

These GUIs are comprised of the following basic items:

- Menu Bar - The menu bar allows the user to access any of the commands applicable to this window at any time. The items available on the menus are shown in Table 3-1:
- Tool Bar - The Tool bar allows the user to select from a subset of the commands.
- Status Bar - The status bar shows the users what is going on in the system. The status bar includes: messages about what the user should do next when they select a specific command, error messages from the server, and information on what editing has taken place at this client and other clients.

Menu Items in Agent Design Window

- File Menu
 - o Save Project
 - o Exit
- Edit Menu
 - o Add Type
 - o Add Subscribe
 - o Add Attribute
 - o Add Message
 - o Remove Type
 - o Remove Subscribe
 - o Remove Attribute
 - o Remove Message
- Edit Control Menu
 - o Request Agent Edit Control
 - o Relinquish Agent Edit Control
- Tools Menu
 - o Check Conversations
 - o Compile Code
 - o Refresh Project
 - o Expand/Collapse Subscription Info
- Help Menu
 - o Getting Started...
 - o Help
 - o About

Menu Items in Conversation Design Window

- Close Menu
 - o Close Window
- Edit Menu
 - o Add Conversation
 - o Delete Conversation
 - o Add State
 - o Delete State
 - o Mark State as Initial
 - o Unmark State as Initial
 - o Mark State as Final
 - o Unmark State as Final
 - o Add Conversation Rule
 - o Delete Conversation Rule
 - o Edit Conversation Rule
- Edit Control Menu
 - o Request Conversation Edit Control
 - o Relinquish Conversation Edit Control
 - o Request Conversation Rule Edit Control
 - o Relinquish Conversation Rule Edit Control
- Help Menu
 - o Getting Started...
 - o Help
 - o About

Table 3-1 - List of JiVE Menu Items

- Editing Canvas - The editing canvas is where the graphical representations of the agents or conversations are located. This is where the user would interact to add, delete, or edit any entities in the design.
- Popup Menu - This is another way for the user to select applicable commands. This menu will show the applicable commands according to where in the Editing Canvas the user has clicked. In Figure 3-3, the user has right clicked on ‘Agent1’ and is presented with several possible commands: Delete Agent, Edit Agent Properties, Edit Conversations, Request Control and Relinquish Control.

3.2.3 Petri-Net Analysis

As was discussed in Section 3.2.1, we specify the conversations and conversation rules of the multi-agent system using a finite state machine representation. This representation allows for Petri net [Peterson 1981] description of the conversations, which then allows for the validation of the design [Chauhan 1997]. The conversion of the finite state machine into a Petri net is as follows [Galan, Baker 1999]:

- First, each state in the automata is represented by a place in the Petri net. This is a one to one mapping of states to places.
- For each conversation rule present in the state machine, a transition is added to the Petri net. Again, this is a one to one mapping of conversation rules to transitions.
- For each conversation rule, add an arc from the place corresponding to the beginning of the conversation rule to the transition.
- Then add another arc from that transition to the place corresponding to the end of the conversation rule.
- The transition must now be enabled by any and all messages, which trigger that conversation rule. (I.e., the transition firing corresponds to the conversation rule receiving its required utterance and executing.) For each conversation rule do the following:
 - Add a place for each utterance that will enable that conversation rule.
 - Add an arc from the transition, corresponding to each utterance that will enable it to the place(s) in the previous step.
 - If there are multiple utterances that will enable a conversation rule, then the arc, transition, and arc corresponding to that conversation rule will need to duplication.
 - Add an arc from the place(s) in the previous step to this conversation rules transition (or duplicated transitions).
- For each final state in each conversation, a transition is added. For each final state, an arc is added to its transition.
- For each conversation, one place is added, and arcs are constructed from each of the transitions in the previous step to the place added in this step. This represents the

corresponding conversation reaching a final state. When the place added in this step contains a token, that conversation has reached a final state.

- One more transition is added for the entire conversation set, which corresponds to all of the conversations reaching a final state and then resetting each conversation to their initial states. An arc is added from each place added in the previous step to the transition added in this step. One arc is added from the transition in this step to each place, which is marked as an initial state, thus resetting the Petri net.
- Finally, one token is added to each place that corresponds to an initial state.

Having built the Petri net, from the original conversation model, we can analyze the multi-agent system for coherency. There are several properties of a Petri net that are required in order to consider that conversation model coherent [Peterson 1981]. The Petri net must be safe: We consider a place safe if, for all possible markings, the number of tokens in that place never exceeds one. Furthermore, we consider the Petri net safe if all of the places in it are safe. The Petri net must also be live: A transition t_j of a marked Petri net is said to be *live* under a marking M_0 if, for all markings $M \in R[M_0]$, there exists a sequence of transition firings which results in a marking that enables t_j . A Petri net is said to be live if all its transitions are live. This implies that the Petri net does not contain any deadlocked states (i.e., a reachable marking in which none of the transitions are enabled).

This Petri net is then analyzed using an external Petri net verification tool [Brink 1996]. The Petri net is analyzed for the following properties: safeness, liveness, and no deadlock conditions. If the Petri net has all of these properties the conversations for that system are considered coherent and logically consistent. If the Petri net does not have one or more of the properties, JiVE will return a message indicating to the user where the problem in the conversation might be [Galan, Baker 1999].

When a user indicates that they would like to run the analysis, the server (*JiVE_Server*) creates an instance of the *PetriNetTranslator*, which receives as a parameter the current state of the design, see Appendix B. In reality, this is just the *Repository* object, which always contains the current state of the multi-agent system design. The *PetriNetTranslator* then creates the Petri net, independent of the external analysis tool, using the algorithm described above. A class that

is dependent on the external analysis tool, called *PetriToolRepresentation*, which implements the *PetriNetRepresentation* interface, facilitates this. *PetriToolRepresentation* encapsulates the representation of the external analysis tool by providing generic methods (as dictated by *PetriNetRepresentation*) to create a Petri net and then converting that Petri net information into the proper format. This construction allows for the replacement of the external analysis tool. A new class implementing the *PetriNetRepresentation* interface could be written for a new analysis tool, and the existing conversion code would not need to be altered.

The design of a multi-agent system in JiVE must also take into account conversations that may never start. The analysis of this aspect of a multi-agent system is unforgiving. If a conversation in the system is intended to be allowed to never execute then that must be explicitly defined. To define this condition, the initial state of the conversation should also be a final state. In the absence of this, the analysis checker will indicate a deadlock condition, wherein the conversation in question never starts and the remaining conversations in the system are finished.

3.2.4 Code Generation

The code generation aspect of JiVE extends the ability of the user to design and test their conversations before writing code by generating the code needed to implement agents and conversations in JAFMAS. JiVE generates Java source code files that match the JAFMAS format. It allows the user to add custom agent code, giving the agent its “behavior”. Even after the user adds custom code to the source code files, the JAFMAS code generator can still regenerate the conversation code. Appendix C gives an example of the generated code.

The *CodeGenerator* is implemented as a static class, which is invoked from the *Repository* class. The *CodeGenerator* takes as parameters the entire database of objects in the system, including agents, conversations, conversation rules, messages, as well as lists of all of the various properties. The *CodeGenerator* systematically goes through each of those objects and places them in the correct section of code.

Any code that might be changed by JiVE when the code is regenerated is placed in between double brackets, like in the following code example:

```
//{{ System defined fields: Do not edit
```

```

    public Conversation conv1 = null;
    //}}

```

Before the *CodeGenerator* regenerates the source code file, it stores the user modified portion of the code (i.e. the code not between the double brackets) and sets it aside to be placed back in the source code in the same place. In this way, the user can manually modify the source code and still be able to regenerate the core agent and conversation code from JiVE (assuming they don't edit inside the brackets).

3.3 System Implementation

This section provides a more detailed discussion of JiVE's implementation.

3.3.1 Architecture

The JiVE application is written in Java, and uses a client server model. The entire JiVE Package consists of a JiVE Server, implemented as a Java application and a JiVE Client, a Java Applet. The server application is designed to run on a machine that has a web server. The web server can serve the JiVE Client applet to users who wish to log in to a JiVE session on a JiVE Server. When a user starts the applet, they will be given the opportunity to log into the *JiVE_Server*. The *JiVE_Server* validates the user's logon, which in turn starts the main agent design window, and gets the initial state of the agent design. The user is now logged on to the JiVE System. As multiple clients log on to JiVE, the multi-agent system can be designed collaboratively and distributedly. The *JiVE_Server* has full access to all of the Server's file

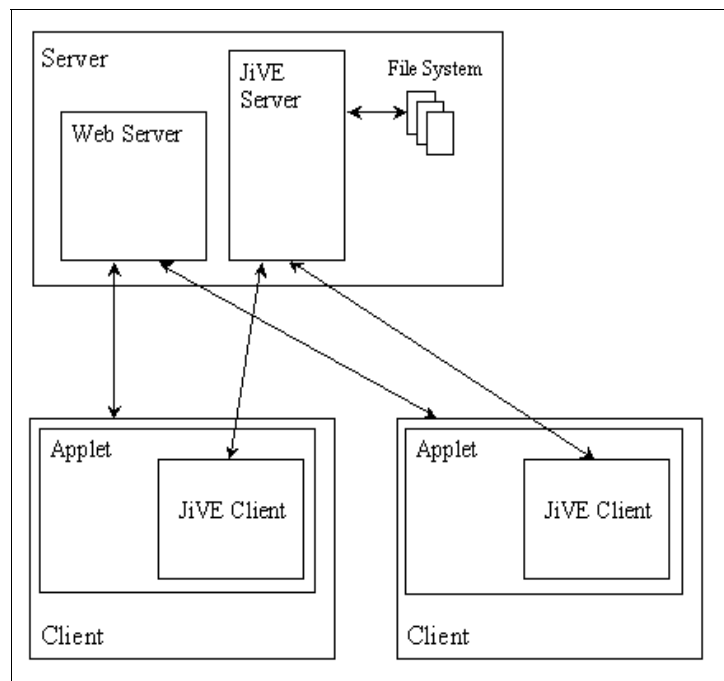


Figure 3-5 - General JiVE Architecture

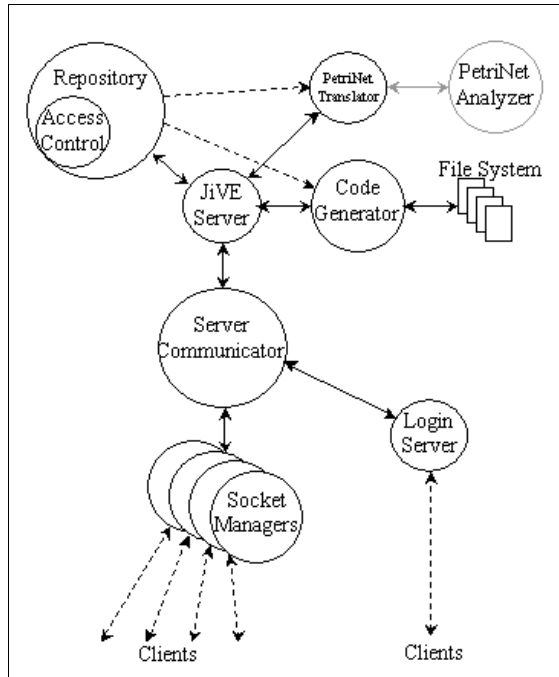


Figure 3-6 - Server Architecture

systems so that, when prompted by a client, the source code can be generated or the project file can be saved for later use. Figure 3-5 illustrates this architecture.

3.3.2 JiVE Server

The JiVE Server is written as a Java application. The JiVE Server running on the web server listens for any clients trying to connect to it. The JiVE Server actually listens on two different ports for two different types of client messages. First, there is a *LoginServer* object, which responds to queries by the various login clients, telling the clients what projects are currently available on the server. (The client also has the option of creating a new

project). Second, the server listens to a port in order to make the TCP/IP Socket connection with a potential client. Upon validating the client's login data, the server puts the socket into an index of active client sockets. See Figure 3-6.

The mechanism that controls the communication between each client and the server, is an object called *SocketManager*. Each client connects to the server through one of these *SocketManager* objects, which is running as an independent thread and listens to the socket connection with the client for a message. Upon receiving a message the socket manager passes the message to the *ServerCommunicator* object for handling.

The *ServerCommunicator* object's tasks include

- Keeping track of the various clients that are connected to the system (via the socket managers)
- Registering new clients, as they log in.
- De-registering clients, as they log out, or if their connection gets broken.
- Processing messages coming into the server and messages going out of the server.

3.3.3 JiVE Client

The JiVE Client is written using the Java 2 Swing components. Swing is the name Sun has given to a set of lightweight GUI components. This includes a lightweight version of many of the Java 1.x GUI components.

The JiVE Client is implemented as a series of windows, which are all descendant from a *JApplet* (The Swing equivalent of an *Applet* object). Figure 3-7 illustrates this architecture. Not shown in the figure are the various dialog boxes used in the client, outlined in Table 3-2.

The entry point to the JiVE Client, the applet, can be opened in a Java Enabled web-browser or by using the JDK appletviewer (as of this writing, the major web-browsers were just beginning to support Java 2).

The JiVE applet does nothing more than create the *JiVE_Client* object. The *JiVE_Client* object opens a *loginWindow* object and creates a *ClientCommunicator* object. Upon creation, the login window

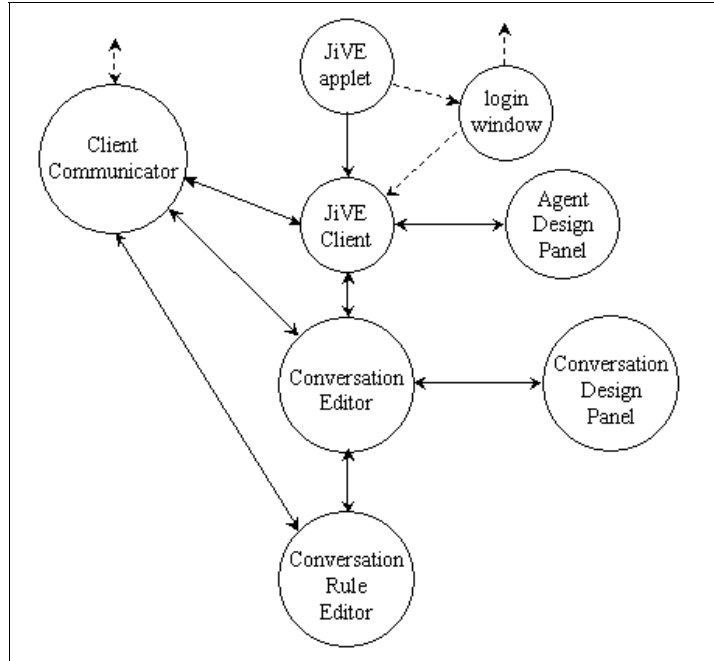


Figure 3-7 - Client Architecture

must query the JiVE Server for the available current projects; it does this using a TCP/IP Socket on a predefined port. The user can then attempt to login to the system, either using an existing project or by creating a new project. The login window attempts to validate the user by sending a message via the *ClientCommunicator* to the JiVE Server, which validates the users name and password. If the login is valid then the JiVE Client opens the Editor window, otherwise the user can retry their login.

The user is now looking at the main editing screen for the JiVE application, and can begin constructing and/or editing the multi-agent system. Chapter 4 provides a comprehensive description of the multi-agent system design process.

The *ClientCommunicator* class handles all of the communication between the client and server, for the client applet. In general the communication that transpires when the user makes any changes to any of the entities in the design goes as follows: The GUI uses the *ClientCommunicator* to send a message to the server. The JiVE_Server handles the editing

- *AboutDialog* - This is a small dialog box that gives information about the program and its author, when opened from the 'About' menu selection.
- *AddPropertyWindow* - This dialog box is used for adding any of the basic agent properties to the system, such as, agent types, attributes and subjects. It is opened from the *JiVE_Client* window.
- *AgentPropertiesWindow* - This dialog box is used for editing the properties of a given agent, either upon creation or for later editing. This is opened from the *JiVE_Client* window.
- *ConversationPropertyEditor* - This dialog is used for adding a conversation and for editing the name of the conversation.
- *ConversationRuleChooser* – This dialog allows the user to indicate which conversation rule they are attempting to select when it is ambiguous.
- *EditMethodsWindow* - This dialog box allows the user to specify the methods to be called by the *suchThat* clause of a conversation rule. It is opened from the *ConversationRuleEditor* window.
- *HelpWindow* - This is a small dialog box that contains help text when the user selects a help button or menu item. This window is opened from various locations in the system
- *InformationWindow* - This dialog box displays a block of text to the user for informational purposes, and is opened from various locations in the GUI.
- *MessageEditorWindow* - This dialog box is opened when the user tries to add or edit a message type in the system. It is opened from the *JiVE_Client* window.
- *RemovePropertyWindow* - This dialog box display a drop down box allowing the user to remove an item from the select property. This is opened from the *JiVE_Client* window
- *StatePropertyEditor* - This dialog opens when the user adds a state to a conversation, allowing the user to specify the name of the new state. It is opened from the *ConversationEditor*.

Table 3-2 - List of Dialog Boxes and Their Functions

change appropriately and responds to the client(s). If the editing change is successful, the server will send a message out to all of the clients notifying them of the change, including the original ‘editor’. If the editing change is not successful, then the server will notify the user who attempted the change of why the change was not successful. The *ClientCommunicator* handles these incoming messages, and passes them to the proper GUI screen for display to the user.

Part of the creation of a new project requires the user to specify the agent communication language that their multi-agent design will use. The default or ‘Standard’ language selection in JiVE utilizes a subset of the JAFMAS *Message* class. This means that the designer must specify their own set of messages for their specific application. JiVE requires the user to specify the type of each message and optionally the ‘intent’ of a message. This represents a subset of the available fields in a JAFMAS *Message*; the user can optionally, manually specify additional message fields by editing the generated source code. The ‘Standard’ choice provides the greatest flexibility.

Besides the ‘Standard’ choice there is the option to utilize one of the predefined agent communication languages built into JiVE, which includes ‘KQML’ or ‘ACL’. The ‘KQML’ and ‘ACL’ language choices preload a set of performatives into the set of available messages. These performatives conform to the standard set of performatives of KQML [Labrou and Finin 1998] and FIPA ACL [FIPA 1997]. To support these Agent Communication Languages there are two message classes included with JiVE that extend the original JAFMAS messaging class. These classes, *KQMLMessage* and *ACLMessage* allow the user to manually fill in the remaining fields (i.e. besides the performative) and then can be used in much the same way as the original JAFMAS *Message* class.

3.3.4 JiVE Communication

The communication between the clients and the server are TCP/IP Socket connections. On the server side, the Socket is originally opened in the *JiVE_Server* class, which passes it to the *ServerCommunicator* who registers the client and creates a *SocketManager* object to listen for any client communication. The *SocketManager* is a threaded object that allows the server to listen to more than one socket, allowing for multiple clients. Upon receiving a message from a

client, the message passes to the *ServerCommunicator* for handling. On the other end of the communication channel, the client uses a similar object, *ClientCommunicator* to handle all messages coming in from the server and going out to the server.

All of the communications between the clients and server utilize the *JiVE_Message* class. The *JiVE_Message* class allows for the storage of key/value pairs that can be sent via the socket (i.e. it implements the Java *Serializable* Interface) and extracted on the other end. The fields that are included in each *JiVE_Message* are extracted in the ‘Communicator’ objects on either end of the communication channel.

Table 3-3 outlines the different messages sent between the clients and the server. The clients only communicate with the server. The server however can communicate with all of the clients. One set of server message types are sent only to the client who originally made the request. These generally are failure messages like ‘Agent Not Added’, that would indicate to the client who tried to add the agent why the request failed, or requests for information. Another set of server message types are sent to all clients currently logged on to the project. These messages are generally successful editing messages, which indicate to all users that there has been a change to the design.

Client Messages to Server	Server Messages Sent to One Client	Server Messages Sent to All Clients
Login	Login Accepted	Agent Added
Save	Login Not Accepted	Agent Deleted
Add Agent	Agent Not Added	Agent Moved
Delete Agent	Agent Not Deleted	Type Added
Move Agent	Agent Not Moved	Subscribe Added
Add Type	Type Not Added	Attribute Added
Add Subscribe	Subscribe Not Added	Message Added
Add Attribute	Attribute Not Added	Type Removed
Add Message	Message Not Added	Subscribe Removed
Remove Type	Type Not Removed	Attribute Removed
Remove Subscribe	Attribute Not Removed	Message Removed
Remove Attribute	Message Not Removed	Agent Edited
Remove Message	Subscription Info. Returned	Conversation Added
Get Subscription Info.	Subscription Info. Not Returned	Conversation Deleted
Get Agent Properties	Return Agent Properties	Conversation Moved
Edit Agent	Agent Properties Not Returned	State Added
Add Conversation	Agent Not Edited	State Deleted
Delete Conversation	Conversation Not Added	State Moved
Move Conversation	Conversation Not Deleted	State Marked As Initial
Get Conversation Prop.	Conversation Not Moved	State Unmarked As Initial
Add State	Return Conversation Prop.	State Marked As Final
Delete State	Conversation Prop. Not Returned	State Unmarked As Final
Move State	State Not Added	Conversation Rule Added
Mark State As Initial	State Not Deleted	Conversation Rule Deleted
Unmark State As Initial	State Not Moved	Conversation Rule Edited
Mark State As Final	State Not Marked As Initial	Return Conversation Analysis
Unmark State As Final	State Not Unmarked As Initial	Relinquish Agent Edit Control
Add Conversation Rule	State Not Marked As Final	Relinquish Conv. Edit Control
Delete Conversation Rule	State Not Unmarked As Final	Relinquish Rule Edit Control
Get Conversation Rule Prop.	Conversation Rule Not Added	Code Generated
Edit Conversation Rule	Conversation Rule Not Deleted	
Check Conversations	Return Conversation Rule Prop.	
Get Agent Edit Control	Conv. Rule Prop. Not Returned	
Get Conversation Edit Control	Conversation Rule Not Edited	
Get Rule Edit Control	Request Agent Edit Control	
Give Agent Edit Control	Request Conv. Edit Control	
Give Conversation Edit Control	Request Rule Edit Control	
Give Rule Edit Control	Code Not Generated	
Refresh Data		
Generate Code		
Close Connection		

Table 3-3 - Message Types

There are, of course, various utility message types such as login messages, save messages, edit control messages, code generation messages, and analysis messages. Table 3-3 also lists these.

3.3.5 Agent Entity Representations

A group of objects that represent the entities in a multi-agent system facilitates the handling of editing and communication within JiVE. These objects are internal to the JiVE system but have a one-to-one correspondence to the entities in the multi-agent system design. The following are represented in the system: Agents, Conversations, Conversation Rules, and Messages. The class names respectively are: *AgentRepresentation*, *ConversationRepresentation*, *ConversationRuleRepresentation* and *MessageRepresentation*.

Each ‘Representation’ has several common properties. First, and most important for JiVE operation, each ‘Representation’ implements the Java *Serializable* interface. This allows these objects to be turned into a serial stream, which can then be sent over a communication channel, in this case the TCP/IP Sockets. This makes sending any information about these agents as simple as writing the object to the stream. Each of these ‘Representations’ also encapsulates the information about how they are written to a file. This is implemented as a *toString()* method in each class.

Additionally, each ‘Representation’ object contains the information about which user, if any, has access control of it. This information is a *User* Object. For example, to check the edit control of an *AgentRepresentation* object called ‘agent’ we can just call a method to find out who currently holds the access control:

```
User userWithControl = agent.getUserWithEditControl();
```

Then we can compare the ‘userWithControl’ object with other *User* objects, to determine who has or will get edit control of that entity.

The agent’s properties, represented in the *AgentRepresentation* object and summarized in Figure 3-8, are as follows:

- Agent Name - The agent’s name
- Attributes - Zero or more attributes that represent the services that this agent can provide to the system.
- Agent Type - The type of the agent.
- Required Resources - 0 or more attributes that this agent needs from other agents in the system

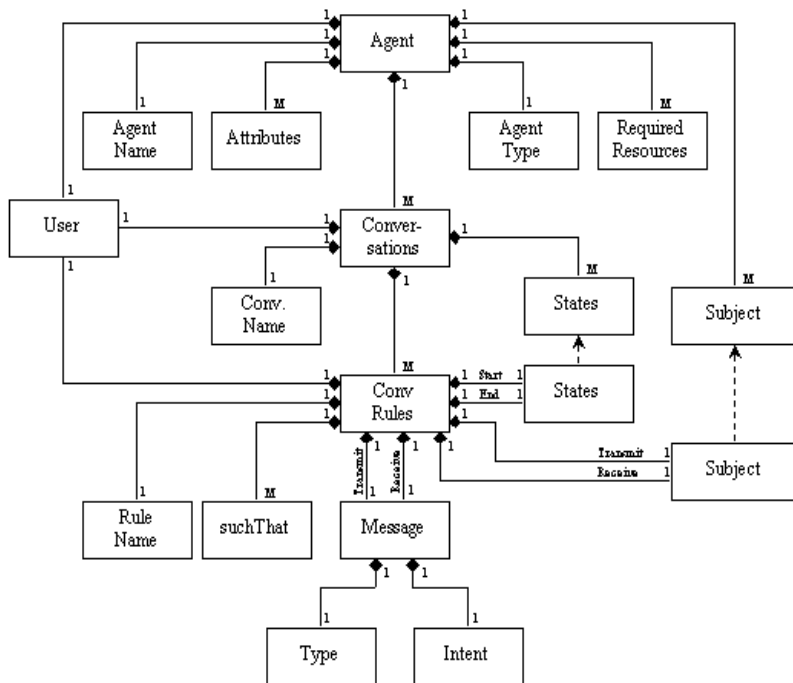


Figure 3-8 - Object Relationship

- Subject - Zero or more multicast channels to which this agent subscribes/publishes.
- User - The user object representing the user in the system who has edit control of this agent.
- Conversations - The conversations of this agent. Represented by *ConversationRepresentation* objects.

The properties of a conversation are represented in the *ConversationRepresentation* object and are:

- Conversation Name - The name of the conversation.
- States - The states in this conversation. The information about which states are initial states and which are final are also encapsulated in this object.
- Users - The user object representing the user in the system who has edit control of this conversation.
- Conversation Rules - A list of the conversation rules that are contained within this conversation. The *ConversationRuleRepresentation* represents each these.

The properties of a conversation rule are represented in the *ConversationRuleRepresentation* object and are as follows:

- Conversation Rule Name - The name of this conversation rule.
- suchThat - A list of the methods to invoke in the *suchThat()* method of the conversation rule. Each of these methods should be defined by the user and should return a Boolean value. After each of these methods returns true, the conversation rule will proceed to the next state.
- Receive Message - A *MessageRepresentation* object for the Message this rule will receive
- Transmit Message - A *MessageRepresentation* object for the Message this rule will transmit.
- Transmit Subscription - The subscription or agent name to which the rule should send a message.
- Receive Subscription - The subscription or agent name from which the rule should receive a message.
- Start state - The starting state of this conversation rule.
- End State - The ending state of this conversation rule.
- Users - The user object representing the user in the system who has edit control of this conversation rule.

MessageRepresentation object has the following fields:

- Type - The type of the message. In JiVE, this is what uniquely identifies the message
- Intent - The intent of the message.

The *MessageRepresentation* object does not need a user object for controlling editing access.

3.4 Summary

This chapter discussed the design and implementation of the JAFMAS integrated Visual Environment (JiVE). This multi-agent system design tool uses an intuitive graphical user interface and supports a collaborative, web-based design environment. Furthermore, JiVE provides automated design validation to check the consistency and coherency of the conversations, as well as source code generation/regeneration. This chapter has also covered, in some cases, specific details regarding the implementation of JiVE. Chapter 4 will provide an example using JiVE to design a multi-agent system.

4. Example

This chapter shows an example of the use of the JiVE design tool. We will walk through the example implementation covering the important aspects of using JiVE. This includes creating a project, designing agents, designing the agent conversations, analyzing the conversations, generating the source code, making modifications to the source code, and deploying the multi-agent system. Shown here is an example multi-agent based implementation of a supply chain. The JiVE User's Manual (Appendix A) presents a detailed description of each specific function in JiVE.

4.1 Supply Chain

The supply chain used in this example is a modified version of the one featured in Chauhan's thesis [Chauhan 1997] which is based on the one presented by Barbuceanu and Fox [Barbuceanu and Fox 1995]. The supply chain consists of a Customer agent, which is the agent at the end of the supply chain, placing the order. The Customer agent places the order with a 'logistics' agent, who must coordinate with one or more 'Plant' agents to provide the required product.

4.2 Project Initialization

The first step to creating the JiVE project is to login to the server and create the project. The project will be called SupplyChain and we will use the standard language (cf. Section 3.3.3).

Figure 4-1 shows the login window. Pressing 'Okay' will verify our User Name and Password. When a user specifies a new project name, the project files and directory structure are created on the server. When a user specifies an existing

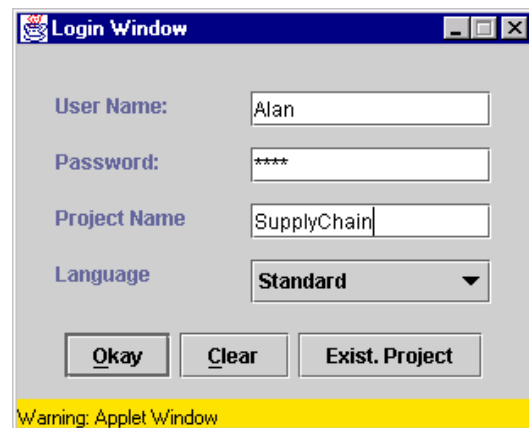


Figure 4-1 - Login Window

project upon login, the project files are loaded from the existing files and the client is initialized with the state of the project.

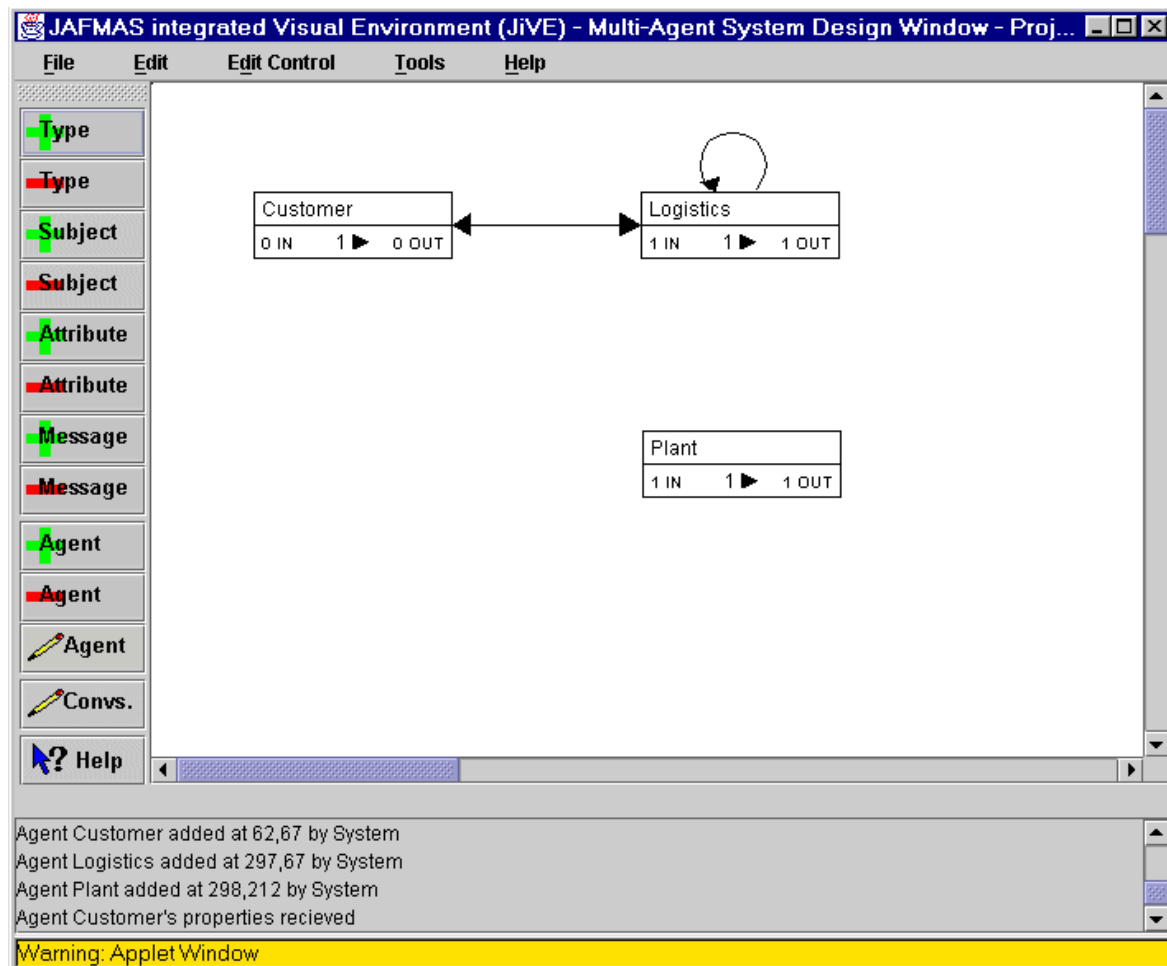


Figure 4-2 - Designing Agents

The first window that appears upon successful login validation is the Multi Agent System Design Window (Figure 4-2). This is where the agents are displayed and edited.

4.3 Designing the Agents

The Multi-Agent System Design Window is where we add the agents and the agent's properties. We start by adding the properties from which the agents will select. These properties are the agent 'Type', agent 'Attributes', and 'Subjects'. To add an agent 'Type' we select 'Add Type'

from the 'Edit' menu or the menu bar, or we click on  on the toolbar. This presents us with a window, where we type the name of the property. Figure 4-3 shows the addition of a 'Customer' type. We proceed to add the remaining types: 'Logistics' and 'Plant'. The 'Attributes' are added in a similar manner: 'Customer', 'Logistics', 'Assembly', 'Paint', and 'Transportation'. We also add the 'Subjects' that the agents can subscribe/publish to, which in this case is just one 'Subject' called 'Factory'.

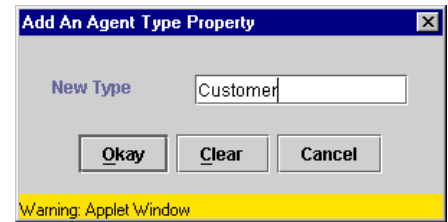
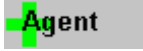
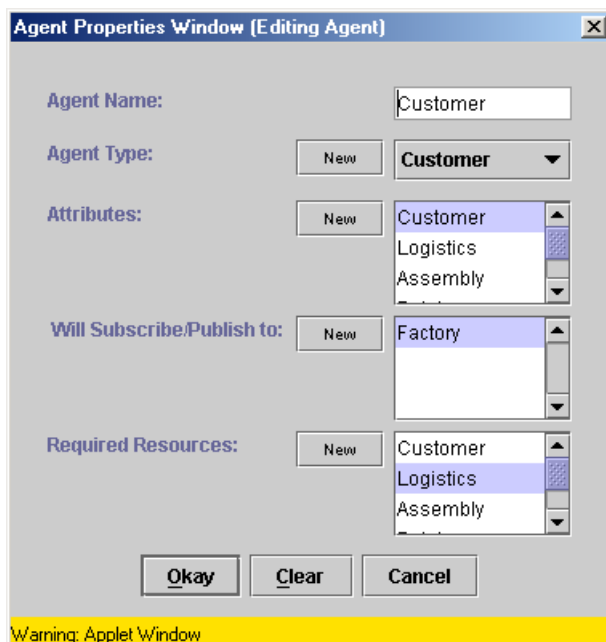


Figure 4-3 - Adding a Property

After adding the various properties, we can begin adding the agents. The three agents that the Supply chain example has are, 'Customer', 'Logistics', and 'Plant'. There are several methods by which we can add the agents, including selecting from the menu bar, from the toolbar or the popup menu.

To add the 'Customer' agent we click on the  toolbar button, and then click on the location in the Agent Design Panel where we want the agent to be placed. The Agent property window appears allowing us to specify the properties of the 'Customer' agent, shown in Figure



4-4. We type in the name of the agent, 'Customer'. The agent is of type 'Customer'. The attributes that this agent has are classified as 'Customer' attributes. The agent denotes the intent to subscribe/publish to the only subject available, which is 'Factory'. We also specify the type of resource this agent requires as 'Logistics'. This means that this agent will look for agents the have the attribute of 'Logistics'. After specifying each of the properties for the agent we select 'Okay'. The 'Add Agent' Request is submitted to the server and we see a message in the status bar

indicating that the agent was added to the design, and the agent appears on the Agent Design Panel.

We repeat this process for the two other agents, 'Logistics' and 'Plant'. Figure 4-2 shows the three agents, as placed in the Supply Chain design. Agents can be dragged around the design window and placed where desired.

Each agent, once created, has a set of conversations associated with it, which must be added to the design. Opening the Conversation Editor for that agent allows for the design and editing of the conversations.

4.4 Designing the Conversations

The Conversation Editor allows for the design and editing of the conversations for a particular

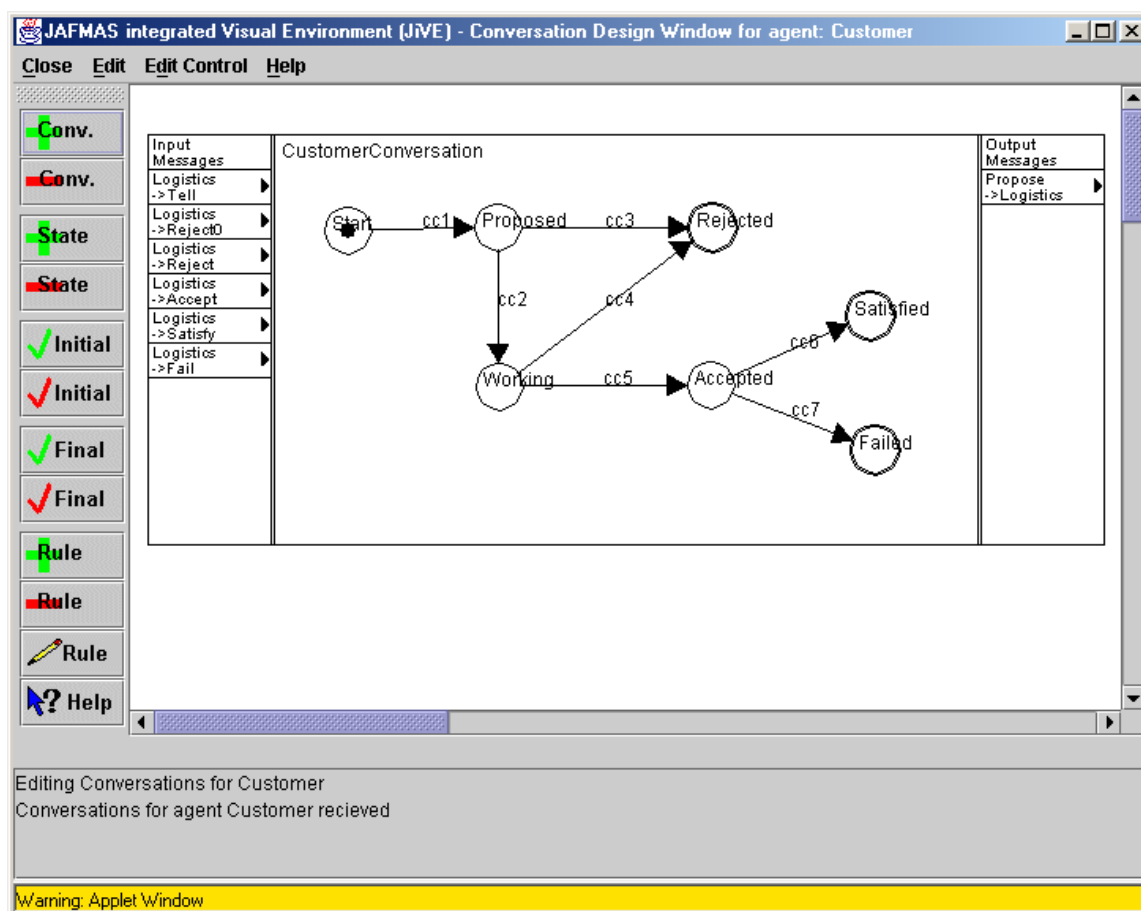


Figure 4-5 - Customer Agent's Conversation Design

agent. Each agent can have one or more conversations, the design panel shows all conversations, with each conversation denoted by a labeled rectangle. To each side of the labeled rectangle are summaries of the input and output messages. Clicking the triangle on the right side of each summary entry will provide a menu with links to each referenced conversation. To add a conversation we select  from the toolbar or 'Add Conversation' from the 'Edit' menu on the menu bar. We click and drag on the Conversation Design Panel, which prompts us to specify the name of the conversation, in the case of Figure 4-5, the name of the conversation is 'CustomerConversation'. With the conversation placed on the panel, we can begin adding the elements of the conversation, including the conversation states and the conversation rules. To add each state in the design select 'Add State' from the menu bar or  from the toolbar and place the state in the conversation. We can specify the name of the state, as well as whether it is an initial state, a final state, both, or neither. These properties can be changed as we progress in the design.

The conversation rules, displayed as an arrow between two states, specify the conditions needed to transition between two states in a conversation as well as the result of that transition. To add the conversation rules, after selecting ‘Add Conversation Rule’ from the menu bar or  from the toolbar, we click on the starting state for the conversation rule followed by a click on the ending state of the conversation rule. The conversation rule appears as a directed line between the start state and the ending state, labeled with the conversation rules name. The conversation rule editor window is shown, where we can specify the conversation rule’s properties.

4.5 Designing the Conversation Rules

Each conversation rule has a set of properties that defines the means by which the conversation

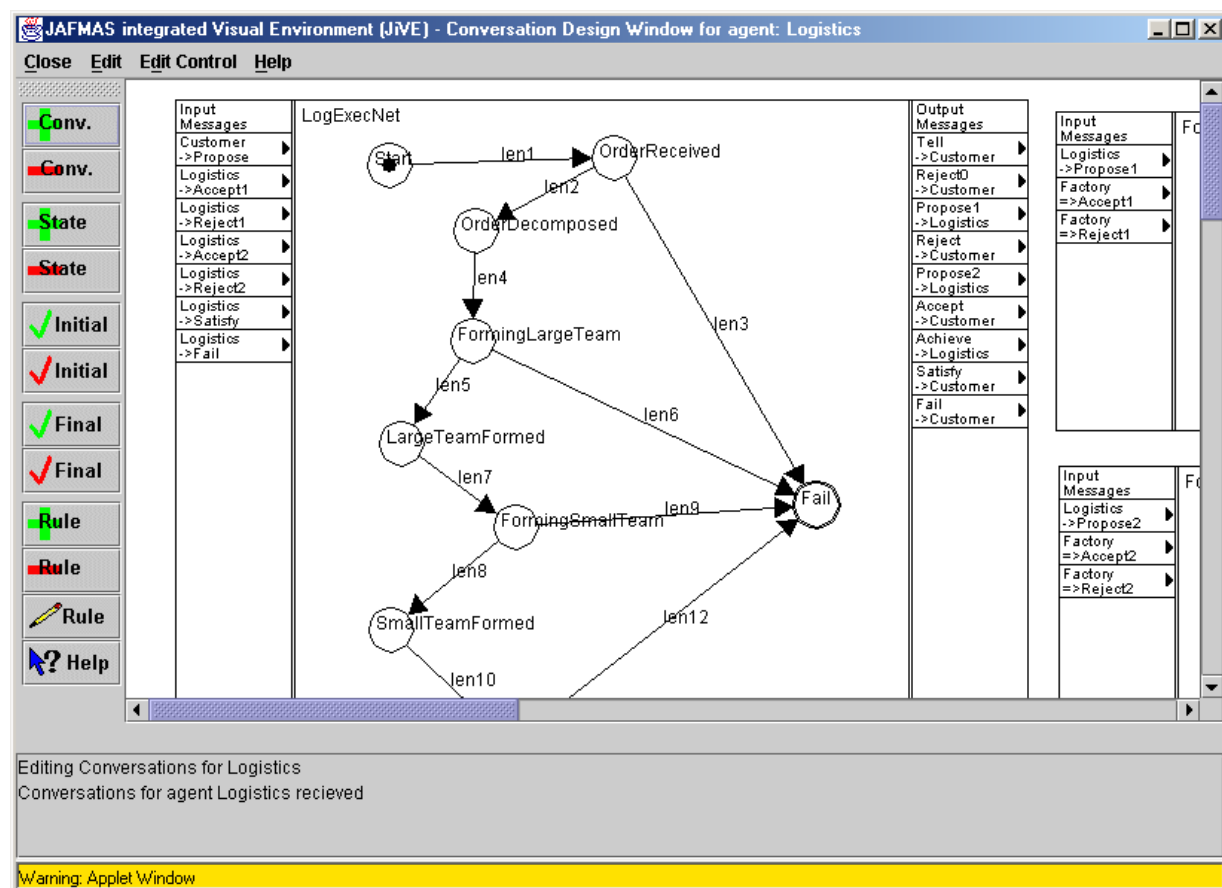


Figure 4-6 - Logistic Agent's Conversation Design

can move from one state to the next. Therefore, the conversation rules substantially specify the behavior of the system. JAFMAS's representation of the order of operation of a conversation rule is as follows:

- The message received which enables this conversation rule, and where it is received from (Whether it is another agent or a subscription channel.)
- doBefore, a method that is executed before the conversation rule is executed.
- A set of methods that must be true in order to proceed, called 'suchThat'.
- doAfter, a method that is executed after the conversation rule is executed.
- The message that is sent out because of this conversation rule being executed and where it is sent.

JAFMAS's doBefore and doAfter methods cannot be specified via the JiVE GUI. The designer manually fills in these methods in the generated source code (c.f. Section 4.8). The design decision to exclude the specification of these two methods in the GUI has two aspects. First, allowing the specification of the code in these methods through the visual environment would preclude the designer from making changes and additions to the code outside of JiVE, due to JiVE's architecture. Second, writing code for just one method, in the visual environment, would not give the designer the needed flexibility to implement cohesive code. Some code is automatically included in these methods, relating to the communications, as specified in JiVE. This code should remain and user defined code can be added to the appropriate part of the method.

This section shows the design of a conversation rule, specifically in the Logistics agent, in the LogExecNet conversation.

We will create the conversation rule named 'len6' which connects the 'FormingLargeTeam' state to the 'Fail' state, shown in Figure 4-6. We see in Figure 4-7 that the name is specified. The enabling message can come from either a subscription channel or a specific agent. Figure 4-8 shows the drop down menu, which lists the possible choices for the sender and/or recipient of a particular message. First are listed the possible subscription channels followed by the agents in the

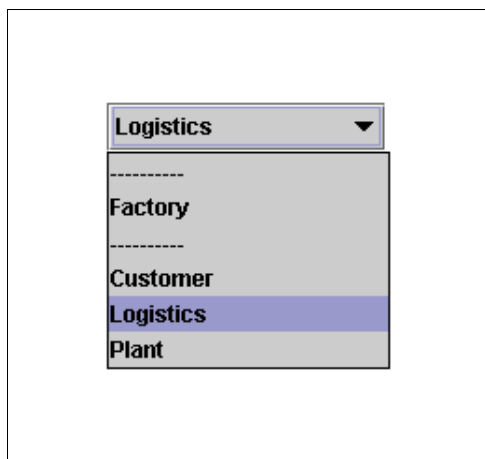


Figure 4-8 – Specification of Message Sender/Recipient.



Figure 4-7 - LogExecNet Conversation Rule Design

system. In this case, we chose for the enabling message to come from The 'Logistics' agent, with a message type of 'Reject1'. The conversation rule sends a message before completion, and this message is sent to the 'Customer' agent with a message type of 'Reject'. Clicking

'Okay' submits this conversation rule to the server, and notification of the success of the operation is shown in the status bar.

In addition, the len6 conversation rule has a 'suchThat' method specified. This means that the specified method(s) must return true before the conversation rule can continue. If there are multiple methods specified, they are evaluated using a Boolean And function. Figure 4-9 shows the editor for the 'suchThat' methods. For each method that we would like to add to the conversation we type it in the text box on the top of the dialog and click add. In the case

of len6 we have added the method 'cannotRankContractors'. We are also able to remove any methods that we do not want by selecting them in the list on the bottom, and clicking remove.

In the Conversation Rule Editor (Figure 4-7) several buttons require additional description. The button labeled 'Subject' opens a Subscription/Publication Information window, which gives visibility of the relationships between each agent and each subject, to assist in selecting the Sender and Addressee properties. This window also indicates to the user those subscription/publication channels that were intended to be used. Additionally on the Conversation Rule Editor, the buttons labeled 'New' allow for the addition of new messages to the system.

The creation of the conversations continues until the entire set of conversations, states and conversation rules have been specified.

When we have designed the entire project, we are ready to check our conversations for errors and then generate the source code for our project.

4.6 Analyzing Conversations and Generating Code

The analysis of the conversations is as easy as making a selection on the menu. During analysis, no changes can be made to the design. In addition, the analysis can take some time depending on the intricacies of the design and the processing power of the machine that is used as the server. When the results are returned an alert will be provided concerning any problems in the design. In our Supply Chain example, we have not marked the Initial state of the conversation

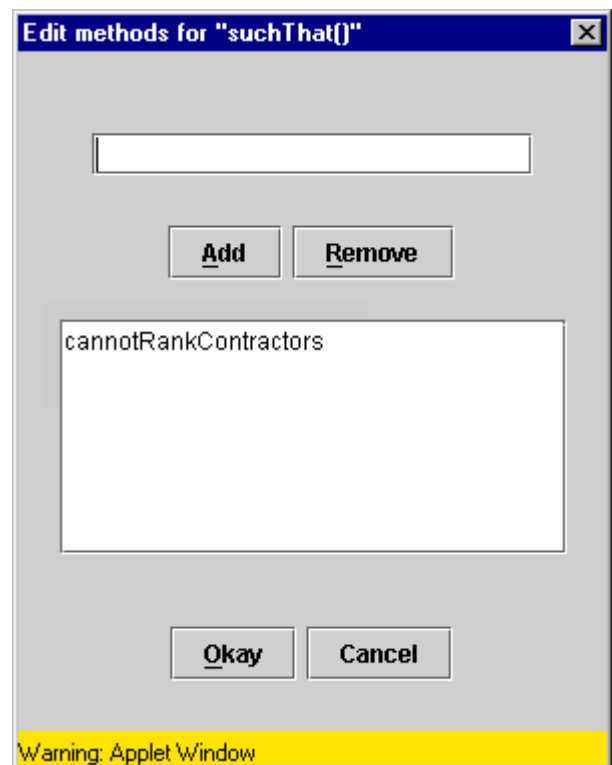


Figure 4-9 - Such That Property Design

‘AnswerFormLargeTeam’ as Final as well. This causes deadlock conditions to appear in the analysis, as shown in Figure 4-10. We then fix the error marking the Initial state of ‘AnswerFormLargeTeam’ as Final as well. And when the analysis is run again it returns a message stating that are conversations are ready to go, which means we can go ahead and generate the source code.

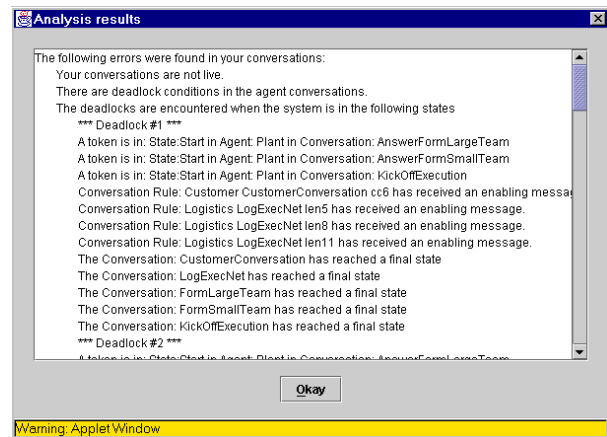


Figure 4-10 - Analysis Results

The generation of the source code requires a click on the appropriate menu, and if successful, the files are stored on the server. The automatically generated source code from the Supply Chain example is given in Appendix C.

4.7 Collaboration

The collaborative features of the design environment are in many cases transparent to the user. That is, the user does not need to worry about edit control for the entities on which they are working. This is a small description of a case where the user would deal with edit control.

Collaboration only comes into play when more than one user is working on the same project at the same time. As mentioned in Section 3.2.1, access control applies to several different entities in the multi-agent system design (i.e. Agent, Conversation, Conversation Rule). For this example in utilizing the collaboration feature we will reference a particular Agent, named ‘Customer’.

In this example, the first user to login to the system has the user name ‘Alan’. ‘Alan’ is the user who created the project, as well as the agent ‘Customer’. When ‘Alan’ created ‘Customer’ that user was automatically given edit control of the agent. ‘Alan’ had the ability to move that agent, edit that agent’s properties, or delete that agent if desired. The next user, named ‘User’ logs into the project that has already been started by ‘Alan’. When ‘User’ logs in, the current system will be initialized and displayed on their client window. ‘User’ may want to move the agent ‘Customer’ that was previously added to the design by ‘Alan’, so ‘User’ tries to move the agent.

'User' receives a message in the status bar (c.f. Figure 3-3) back from the server indicating that the move operation was not successful because 'User' does not have edit control. At this point, if 'User' wants to move the agent, they must request the edit control of 'Customer'.

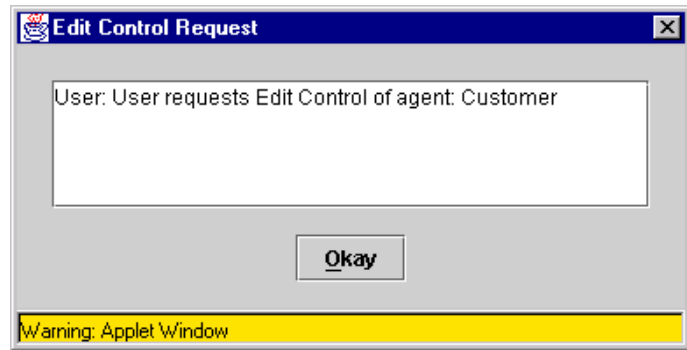


Figure 4-11 - Edit Control Request Dialog Box

They do this by selecting 'Request Edit Control of Agent' from the 'Edit Control' menu on the menu bar and clicking on 'Customer' (or selecting it from the popup menu.) This request is sent via the Server and 'Alan' sees a pop up box as shown in Figure 4-11, saying the Edit Control of the agent 'Customer' is requested by 'User'. 'Alan', can then select 'Relinquish Edit Control' from the popup menu, or select it from the 'Edit Control' menu on the menu bar and click on the 'Customer' agent. All users in the system will see a message in their status bar, stating that edit control of 'Customer' has been relinquished. 'User' is now free to move the 'Customer' agent, and upon moving it will automatically receive edit control.

4.8 Using the Design

To generate the source code for the multi-agent system design, select 'Compile Code' from the 'Tools' menu on the menu bar. The source code is generated and placed in the project directory (see Appendix A.1.2). The source code generated by JiVE based on the design in our project needs several additions before it can be deployed as a multi-agent system. First, it is important to note that even after making changes or modifications to certain areas of the source code it is still possible to change the conversation designs and regenerate the source code. There are lines in the source code that look like this:

```
//{{  
    JiVE Generated Code is here.  
//}}
```

These lines delimit the area of the source code, which is generated by JiVE. As long as the user does not add or edit any text within those areas, their code will remain intact.

The first thing that needs to be added to the source code before we can even compile it, are any methods we specified in the ‘suchThat’ property. The different methods can be located by compiling the source code, which will generate an error message pointing to each ‘suchThat’ method that has not been elaborated. These methods should return a Boolean value and can take no parameters. There are no restrictions on what these methods can contain. They should however indicate, by returning true or false, whether a conversation that checks its status can complete its transition.

The methods `doBefore()` and `doAfter()`, which are included in the automatically generated source code, can be filled in if desired by the user. This would be the location to do any operation or call any methods that should be done before or after the operation of the conversation rule. These methods can be found in the classes for each conversation rule. In addition, supplemental code can be added to the agents and conversations to provide additional behavior to the multi-agent system.

Finally, once any necessary modifications and additions have been made to the design the system can be deployed. The details on compiling and running the agents are included in the JiVE Users Manual in Appendix A.

4.9 Summary

This chapter has presented an example design session using JiVE. Each aspect of designing a multi-system was explored and described. Appendix A, the JiVE User Manual, provides more details on the specific commands and their use.

5. Discussion and Future Work

This thesis presented a graphical multi-agent design tool. This chapter will discuss the presented work, and suggest future improvements and enhancements.

5.1 Summary

This thesis presented the design and implementation of a design tool for the development of multi-agent systems. JiVE provides an easy to use interface for drawing the multi-agent system, including visibility of the agent conversations and conversation rules. The coherency and coordination of the set of agent conversations can be verified, using JiVE's PetriNet translator and an external Petri net analyzer.

JiVE collaboration support further enhances the design process by allowing users in different locations to work on the same project. Furthermore, JiVE is platform independent by using the Java™ programming language.

Finally, the generation of the source code in JAFMAS format provides the designer with a starting point for a full-featured multi-agent system.

5.2 Discussion

The design of the supply chain example, presented in Chapter 4, brought forth some interesting results with regard to the design of a multi-agent system that will be checked for coherency. The Petri net analysis provides very strict coherency checking. The downside is that it is not very forgiving in terms of isolating design flaws. The best approach to designing a system in JiVE, assuming the analysis tool is used, is to build the design incrementally. Starting with a smaller design that is known to be coherent and then adding incrementally will expedite the process. Creating an entire project and then trying to use the analysis tool can cause the analysis to never terminate, if there are numerous errors.

The use of the supply chain example as stated in the JAFMAS thesis [Chauhan 1997] highlights another area where accommodations must be made for the analysis tool. The analysis of the agent conversations with programmed constructs (e.g. `suchThat` methods) is often not possible

in JiVE. JiVE's analysis can only capture information about conversation interdependencies based on the explicit communication (i.e. messages). The incorporation of additional analysis tools and formal methods may provide this added analytic flexibility (c.f. Section 5.3.3). At this point, systems designed in JiVE, assuming the analysis tool is used, must refrain from defining the conversations in terms of programmed constructs. In addition to the inability to analyze programmed constructs in the `suchThat` methods, users must be cautious in their use of the `doBefore` and `doAfter` methods, so as not to introduce programmatic control of the conversations.

Finally, the growth of the Petri net is an issue that may be prohibitive for some larger designs. The Supply chain example appeared to be on the edge of what could be analyzed in JiVE. This could be because of the use of a shareware Petri-net analysis tool (Petri Tool) [Brink 1996], where optimizations in speed are still possible. The designer should be aware that the size of the Petri net associated with a particular multi-agent system design can grow large very quickly, and thus drag analysis to a halt. This is especially true for certain constructs such as messages that are sent to multiple agents simultaneously. Given more computing power and improved Petri net analysis tools, this issue does not seem to invalidate the general usefulness of Petri-net tools in evaluating the coherency of multi-agent systems.

5.3 Future Work and Enhancements

5.3.1 Improved Collaboration features

Several features would add value to the collaboration features of JiVE. A white board and chat feature, in which the various users logged on to the design environment could communicate, would be helpful. Examples of this can be found in [Inamdar 1997] and [Chen 1998]. The users could more effectively collaborate by drawing pictures and writing text directly to each other, in tandem with their design efforts.

A feature that would go hand in hand with this would be improved visibility of the users logged in to the project [Inamdar 1997, Chen 1998]. As it stands now, the users logged on to a JiVE

project do not have access to a comprehensive list of who else is currently working on the project.

As previously mentioned, the addition of an Undo feature as found in Joint EMACS [Ressel, Nitssche-Ruhland and Gunzenhäuser 1996] would greatly enhance the usability of the collaborative environment, allowing the users to back out mistakes and /or errors while keeping the design consistent.

5.3.2 Better Graphical Interface

There is additional room for improvement and enhancement in some of the graphical aspects of JiVE. The look and feel of the agents and conversations on the Design Panels could be improved in several ways. First, automatic positioning of the entities on the design panels (e.g. agents, states, conversations) would enable the designer to more clearly see what is already in the system without having to reposition the entities themselves.

The conversation rules positioning are currently straight lines between the start and end state. The look and feel of the conversation design panel could be improved by better routing of these lines. Some examples of this are: to not allow two conversation rules to overlap, or allowing the user to route the conversation rule manually using more than one vertex. This improved routing of the conversation rules would also open up improvement to the conversation rule selection model. Currently a user must select the starting state and ending state of any conversation rule that they would like to add, edit or delete. One click could be removed for editing and deleting, this would make JiVE easier to use.

5.3.3 Improved Efficiency During Design

The efficiency of using this multi-agent system design tool could be improved by adding the following features. First, an ability to reuse conversations, or parts thereof, that have already been designed. Ideally, this would be a copy and paste operation where the user could select a conversation, indicate a cut, and reuse that conversation in another agent's conversations. This would be especially useful in designs where different agents have similar conversations. In addition, the ability to create nested conversation structures would greatly improve the ability to

create a multi-agent system in JiVE. JAFMAS has facilities to support a nested conversation model but these are not supported in JiVE, therefore the designer must manually incorporate nested conversations.

Providing mechanisms for utilizing inheritance would also greatly improve the JiVE design experience. This could include agents as well as agent conversations. Inheritance would allow the user to design a generalized parent agent (or conversation) with properties applicable to an entire set of descendant agents (or conversations). The descendant agents (or conversations) would add additional specific functionality while maintaining the parent's functionality. For example, in a supply chain system, one might have several different types of 'Plant' agents. The specification of each of those 'Plant' agents would be faster if a generalized 'Plant' agent could be specified, and then various additional agents that inherit the 'Plant' agent's properties are specified. A 'Milling' agent, a 'Drilling' agent, and a 'Painting' agent could be added with little additional effort.

Another potential place for enhancement is in the specification of the 'suchThat' property. In the current implementation the user specifies a set of method names that will be joined by a Boolean 'And' operation in the generated source code. Allowing the user to specify an extended Boolean function using method names and the various Boolean operations, would improve the usefulness of this property.

The need to edit the JiVE generated code outside of the JiVE environment could be seen as a weakness of this tool. To overcome this, the incorporation of the editing into JiVE, much like commercial IDEs would provide a fuller user experience. The agent code could then be fully generated by JiVE and even automatically deployed.

5.3.4 Better Project Support

For simplicity, the current design limits the number of actively edited projects to one. In order to allow multiple projects JiVE would need to maintain and control multiple instances of the JiVE Server. In addition, management of the communication between the clients and the multiple servers would need to be added to the server. The ability for multiple projects to be active in the system would increase the usability of JiVE, by allowing for an organization to

place the JiVE server in one central location and allow editing on any project at any time. Additionally, more comprehensive user management features would go along with this very nicely. This could include interfaces for an administrator to manage the JiVE users and the ability to limit user access to certain projects, as is done with the Administration client in TeamRooms [Roseman and Greenberg 1996].

5.3.5 Improved Analysis

The performance of the Petri net analysis, especially given certain constructs, (c.f Section 5.2) could be improved. This could be from the integration of better and more powerful Petri net analysis tools, as well as supplementation by other formal methods. For example, the incorporation of a Z specification language analysis could allow for the incorporate programmatically controlled agent conversation while still allowing the analysis of the coherency of the system as a whole.

5.3.6 Improved ACL Support

The current support in JiVE for the ACLs is limited to allowing the designer to specify a set of standard performatives for the system communication. Support for ACLs could be enhanced allowing for adding standard ACL protocols and additional coherency checking.

In certain ACL specifications there are standard protocols defined, for example FIPA ACL [FIPA 1997]. These protocols provide functionality such as, query, contract-net, and English auction. The ability to add these and other pre-defined protocols to a multi-agent design in JiVE would greatly decrease the design time and improve the functionality.

The coherency checking, outlined in this thesis (cf. section 3.2.3) could be supplemented by checking the correctness of the use of the ACLs. For example, if the user has specified for an agent to receive a message with the performative of ‘Ask’ it could be verified that the agent responds with ‘Tell’. These defined semantics of the ACL provide a guide for how the agents communicate and would be useful as an additional coherency check.

5.3.7 Improvements in Agent Design

JiVE has been closely coupled with JAFMAS, which provides a basis and a framework for implementing multi-agent systems. Although this has affected the JiVE implementation in content and structure, it would be possible to move to a more general approach where the user would have the choice of various agent frameworks to design against. In practical terms for the current implementation, this could mean the replacement of the Code Generator portion of the software with an analogous generator for a different framework.

The design and implementation of a multi-agent system with the validation of the coherency of the conversations, all using a graphical interface has significant value for a designer. Even more value could be added to the design cycle by real-time debugging of the agent system. For example, after designing and checking the system and generating and implementing the source code, the user could then run the system and watch the activity in the agent conversations in the same or similar interfaces in which they were designed.

5.4 Summary

JiVE is presented in this thesis as a tool for the design and implementation of multi-agent systems. JiVE has several aspects that make it an exceedingly useful multi-agent system design tool. JiVE consists of a set of intuitive graphical interfaces, allowing a designer to handily implement a multi-agent system. The interfaces, including the agent design window and the conversation design window, provide a glimpse of the multi-agent system at varying levels of detail. The built in collaboration features of JiVE provide the added flexibility of designing the multi-agent system in a group. This group of designers need not be located in the same room or even in the same building so long as they have a connection to the Internet. The group design process is enhanced, not hindered, by the access control scheme utilized in JiVE, allowing for the almost transparent sharing of a common design area, and the assurance of consistent project data.

JiVE's design validation tool incorporates a unique approach to the verification of multi-agent system communication. The design of the multi-agent system communications in the finite-state machine format and the algorithm for conversion to a Petri-net allow the designer to analyze

their design before deployment. The accompanying direction for the resolution of design errors assists the designer in locating and fixing problems found in the analysis.

The design process is completed by allowing the generation of the source code in JAFMAS format, allowing for the rapid deployment of the multi-agent system. The ability to add additional code to the system design and to subsequently regenerate the JiVE generated portion of the system allows designers to iteratively design their multi-agent system. This entire design process was exhibited with the design of a multi-agent system supply chain, which showed the ease of use and versatile features.

Overall, JiVE is shown as a full-featured multi-agent system design tool, which provides a quick and easy bridge between design and implementation.

Bibliography

- [Aknine and Aknine 1999] S. Aknine and H. Aknine. "Contribution of a Multi-agent Cooperation Model in a Hospital Environment." *Proceedings of the Third Annual Conference on Autonomous Agents*, Seattle, WA, May 1-5, 1999, 406-407.
- [Barbuceanu and Fox 1995] M. Barbuceanu and M. S. Fox. "COOL: A Language for Describing Coordination in Multi Agent Systems." *Proceedings of the First International Conference on Multi-Agent Systems*, July 12-14, 1995, San Francisco, CA.
- [Bellifemine, Poggi, and Rimmassa 1999] Bellifemine, F., Poggi, A., and Rimmassa, G., "JADE – A FIPA-compliant agent framework." *CSELT internal technical report*. Also published in part in the *Proceedings of PAAM '99*, April 1999, pp. 97-108.
- [Bölöni 1998] Bölöni, L. "Bond Objects – a white paper." *Technical report CSD-TR #98-002*, Purdue University, Department of Computer Science, February 1998.
- [Bölöni and Marinescu 1999] Bölöni, L and Marinescu D. C. "A Multi-Plane State Machine Agent Model." *Technical Report CSD-TR #99-027*, Purdue University, Department of Computer Science, September 1999.
- [Bradshaw et al. 1997] J. M. Bradshaw, S. Dufield, P. Benoit and J. D. Woolley, "KAoS: Toward An Industrial-Strength Open Agent Architecture." *Software Agents*, Ed. J. M. Bradshaw, MIT Press, Cambridge, MA, 1997, 375-418.
- [Brandenburg et al. 1998] J. Brandenburg, B. Byerly, T. Dobridge, J. Lin, D. Rajan, and T. Roscoe, "Artefact: A Framework for Low-Overhead Web-Base Collaborative Systems." *Proceedings of ACM 1998 Conference on Computer Supported Cooperative Work*, November 14-18, 1998, Seattle, WA, 189-196.
- [Brink 1996] R. S. Brink, "A Petri Net Design, Simulation, and Verification Tool." MS Thesis. Department of Computer Engineering, Rochester Institute of Technology, September, 1996. (Available at <http://www.csh.rit.edu/~rick/thesis/thesis.html>)
- [Chauhan 1997] D. Chauhan, "JAFMAS: A Java-Based Agent Framework for Multi-Agent Systems Development and Implementation." Masters Thesis. ECECS Department, University of Cincinnati, July 16, 1997. (Available at <http://www.ececs.uc.edu/~abaker/JAFMAS/index.html>)

- [Chen 1998] Q. Chen, "JSIM: A collaborative Environment For Web-based Control System Design." Masters Thesis. ECECS Department, University of Cincinnati, March 1, 1998.
- [Cost et al. 1999] R. S. Cost, Y. Chen, T. Finin, Y. Labrou and Y. Peng, "Modeling Agent Conversations with Colored Petri Nets." *Workshop on Specifying and Implementing Conversation Policies of the Third Annual Conference on Autonomous Agents*, Seattle, WA, May 1, 1999. 59-66.
- [David and Kraus 2000] E. David and S. Kraus, "Agents for Information Broadcasting." *Intelligent Agents VI (LNAI Volume 1757)*, Eds. N. R. Jennings and Y. Lesperance, Springer-Verlag: Berlin, Germany, 2000. 91-105.
- [d'Inverno, Kinny and Luck 1998] M. d'Inverno, D. Kinny and M. Luck. "Interaction Protocols in Agentis." *Proceedings of the Third International Conference on Multi-Agent Systems*, Paris, France, July 3-7, 1998. 112-119.
- [Ehrlich 1999] K. Ehrlich, "Chapter 1: Designing Groupware Applications: A Work-Centered Design Approach." *Computer Supported Co-operative Work*, Ed. M. Beaudouin-Lafon, John Wiley & Sons, West Sussex, England, 1999. 1-28.
- [FIPA 1997] FIPA, "Part 2: Agent Communication Language." *FIPA 97 Specification*, Foundation for Intelligent Physical Agents, Geneva, Switzerland, November 28, 1997. (Available at <http://drogo.cselt.stet.it/fipa/f7a12.zip>)
- [Galan and Baker 1999] A. Galan and A. D. Baker, "Multi-Agent Communication in JAFMAS." *Workshop on Specifying and Implementing Conversation Policies of the Third Annual Conference on Autonomous Agents*, Seattle, WA, May 1, 1999. 67-70.
- [Graham and Decker 1999] J. R. Graham and K. S. Decker, "Towards a Distributed, Environment-Centered Agent Framework." *Intelligent Agents VI (LNAI Volume 1757)*, Eds. N. R. Jennings and Y. Lesperance, Springer-Verlag: Berlin, Germany, 2000. 290-304.
- [Greenberg and Roseman 1999] S. Greenberg, and M. Roseman, "Chapter 6: Groupware Toolkits for Synchronous Work." *Computer Supported Co-operative Work*, Ed. M. Beaudouin-Lafon, John Wiley & Sons, West Sussex, England, 1999. 135-168.

- [Hughes et al. 1996] J. Hughes, J. O'Brien, M. Rouncefield and T. Rodden, "They're Supposed to Be Fixing it: Requirements and System Redesign." *CSCW Requirements and Evaluation*, Ed. P. Thomas, Springer-Verlag: London, 1996. 21-38.
- [Huhns and Singh 1998] M. N. Huhns and M. P. Singh, "Chapter 1: Agents and Multiagent systems: Themes, Approaches, and Challenges." *Readings in Agents*, Eds. M. N. Huhns and M. P. Singh, Morgan Kaufman Publishers, 1998, 1-23.
- [Inamdar 1997] P. C. Inamdar. "Towards Developing Web Browser Based Collaborative Work Environments: The JPECS Experience." Masters Thesis. ECECS Department, University of Cincinnati, March 20, 1997.
- [Jacomino, Geszychter, and Rabelo 2000] Jacomino, A., Geszychter, M. B., and Rabelo, R. J., "MASSIVE Kit for the development of multi-agent systems." Federal University of Santa Catarina, April 2000.
- [Labrou and Finin 1998] Y. Labrou and T. Finin, "Semantics for an agent communication language." *Intelligent Agents IV (LNAI Volume 1365)*, Eds. M. P. Singh, A. S. Rao, and M. J. Woolridge, Springer-Verlag: Berlin, Germany, 1998. 209-215.
- [Mehra and Nissen 1998] A. Mehra and M. E. Nissen, "Case Study: Intelligent Software Supply Chain Agents Using ADE." *Proceedings of the AAAI Workshop on Software Tools for Developing Agents*, 1998. 53-62.
- [Mills 1998] A. Mills, *Collaborative Engineering and the Internet*. Society of Manufacturing Engineers, Dearborn, Michigan, 1998.
- [Moldt and Wienberg 1997] D. Moldt and F. Wienberg, "Multi-Agent-Systems Based on Coloured Petri Nets." *Proceedings of the 18th International Conference on the Application and Theory of Petri Nets*, Eds. P. Azema and G. Balbo, Toulouse, France, June 23-27, 1997. 82-101.
- [Niemeyer and Peck 1997] P. Niemeyer and J. Peck. *Exploring Java, Second Edition*. O'Reilly & Associates, Inc. Sebastopol, CA, 1997.
- [Nissen and Mehra 1999] M. E. Nissen and A. Mehra, "Some Intelligent Software Supply Chain Agents." *Proceedings of the Third Annual Conference on Autonomous Agents*, Seattle, WA, May 1-5, 1999, 374-375.

- [Nwana et al. 1999] H. Nwana, D. Ndumu, L. Lee and J. Collis, “ZEUS: A Toolkit for Building Distributed Multi-Agent Systems.” *Applied Artificial Intelligence Journal*, Vol. 13 (1), 1999, 129-186.
- [Parunak 1996] H. V. D. Parunak, “Visualizing Agent Conversations: Using Enhanced Dooley Graphs for Agent Design and Analysis.” *Proceedings of the Second International Conference on Multi-Agent Systems (ICMAS '96)*, Kyoto, Japan, December 10-13, 1996.
- [Patterson, Day and Kucan 1996] J. F. Patterson, M. Day, and J. Kucan, “Notification servers for synchronous groupware.” *Proceedings of the ACM CSCW '96 Conference on Computer Supported Cooperative Work*, Boston, MA, November 16-20, 1996. 122-129.
- [Peterson 1981] J. L. Peterson, *Petri Net Theory and the Modeling of Systems*. Prentice Hall, Englewood Cliffs, New Jersey, 1981.
- [Prakash 1999] A. Prakash, “Chapter 5: Group Editors.” *Computer Supported Co-operative Work*, Ed. M. Beaudouin-Lafon, John Wiley & Sons, West Sussex, England, 1999. 103-133.
- [Ressel, Nitssche-Ruhland and Gunzenhäuser 1996] M. Ressel, D. Nitsche-Ruhland and R. Gunzenhäuser, “An Integrating, Transformation-Oriented Approach to Concurrency Control and Undo in GroupEditors.” *Proceedings of ACM 1996 Conference on Computer Supported Cooperative Work*, Boston, MA, November 16-20, 1996. 288-297.
- [Reticular 1999a] Reticular Systems, “AgentBuilder: An Integrated Toolkit for Constructing Intelligent Software Agents.” Reticular Systems, Inc., February 18, 1999. (Available at http://www.agentbuilder.com/Documentation/white_paper_r1_3.pdf)
- [Reticular 1999b] Reticular Systems, “Using Intelligent Agents to Implement an Electronic Auction for Buying and Selling Electric Power.” Reticular Systems, Inc. August 30, 1999. (Available at <http://www.agentbuilder.com/Documentation/EPRI/epri.pdf>)
- [Roseman and Greenberg 1996] M. Roseman and S. Greenberg, “TeamRooms: Network Places for Collaboration.” *Proceedings of ACM 1996 Conference on Computer Supported Cooperative Work*, Boston MA, November 16-20, 1996, 325-333.

[Ygge and Akkermans 1997] F. Ygge and H. Akkermans, “Making a Case for Multi-Agent Systems”, *Multi-Agent Rationality: 8th European Workshop on Modeling Autonomous Agents in a Multi-Agent World (MAAMAW '97)*, Ronneby, Sweden, May 13-16, 1997, Eds. M. Boman and W. Van de Velde, Springer 1997.

Appendix A

JiVE Users Manual

This appendix contains installation procedures and directions for using JiVE. There are instructions for starting the client and server portions of the software, as well as instructions for using the client's Graphical User Interfaces.

A.1 Installation

The JiVE software can be installed in several different ways depending on the specific system and intended use. This section first discusses the minimum system requirements, the JiVE file structures, and then the three configuration options.

A.1.1 Minimum System Requirements

The minimum systems requirements for the server and the client are slightly different. The server is written such that it is compatible with the Java Development Kit Version (JDK) 1.1.x. The server needs a JDK 1.1.x runtime engine, and optionally a web server and/or Internet or network connection. The client classes use the Java Swing package, which is part of Java 2 (i.e. JDK 1.2). The client is an applet and therefore needs to be able to run a Java 2 applet that is either a Java 2 appletviewer or a Java 2 compatible web-browser (as of this writing, the major web-browsers were just beginning to support Java 2.)

The processing power of the machines used to run both the client and the server can become an issue for certain older machines. Although less processing power will not generally cause a problem in using JiVE, there are situations where ample processing power is required.

Specifically the analysis performed at the server can be especially computationally intensive. Here the size and complexity of the design strongly effects speed.

Sections A.1.3 through A.1.5 discuss the three installation options for JiVE. They refer to the client files and the server files, listed in Table A-1.

Client Files		Server Files
AboutDialog AddPropertyWindow AgentDesignPanel AgentPropertiesEditor AgentRepresentation ClientCommunicator ConversationDesignPanel ConversationEditor ConversationPropertyEditor ConversationRepresentation ConversationRuleChooser ConversationRuleEditor ConversationRuleRepresentation EditMethodsWindow HelpWindow InformationWindow	JiVE JiVE_Client JiVE_Constants JiVE_Message LoginWindow MessageEditorWindow MessageRepresentation RemovePropertyWindow StatePropertyEditor User Images/*.gif	AgentRepresentation CodeGenerator ConversationRepresentation ConversationRuleRepresentation JiVE_Constants JiVE_Message JiVE_Server LoginServer MessageRepresentation PetriNetRepresentation PetriNetTranslator PetriToolRepresentation Repository ServerCommunicator SocketManager User Users.txt index.html All files in PetriTool directory The JiVE Jar file

Table A-1 - List of JiVE Classes.

A.1.2 File Structure

Figure A-1 shows the JiVE directory structure. The JiVE distribution zip file will create this directory structure in the installation directory, when extracted. The installation directory contains a directory with the complete set of javadoc-generated documentation (docs). The directory containing all of the JiVE Source Code and class files is called JiVE and has a child directory that contains the images used in the JiVE Client. The directory containing the source and class files for the Petri net analysis tool (PetriTool) is also a sub-directory of the installation directory. There is also a directory created that contains all of the individual project

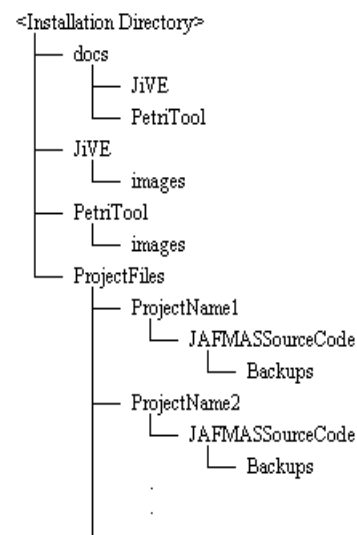


Figure A-1 - Directory Structure

files, including the files used by JiVE, and the JAFMAS Source code generated for each project.

A.1.3 Standard Configuration

The standard configuration of JiVE assumes access to a web server and the desire to utilize the collaborative features. There are instances where it may not be possible to use this configuration due to the specific web server, in these cases follow the directions in Section A.1.5.

Install the JiVE directory structure into a directory that can be accessed via a web server. Once the files are in place the *JiVE_Server* can be started (c.f. Section A.2.1), and users will be able to access the JiVE *index.html* file (which is in the JiVE installation directory). This starts the *JiVE_Client* in the web browser and allows for login to a project (c.f. Sections A.2A.3).

A.1.4 Free Standing Configuration

This configuration allows the use of JiVE if there is no access to a web server. This configuration will not allow the use of the collaborative features. Install the JiVE class files in any directory location and run the *JiVE_Server* and the *JiVE_Client* on the local machine, from the JiVE installation directory (c.f. Sections A.2 and A.3).

A.1.5 TinyHttpd Configuration

If the configuration of the systems web server is such that JiVE directories cannot be accessed, the TinyHttpd web server can be installed to serve up the JiVE client files. The TinyHttpd class is included with JiVE [Niemeyer and Peck, 1997]. Please note that this web server has no built in protection mechanisms. Use at your own risk. Directions to start this web server are in Section A.2.1.

A.2 Server

A.2.1 Running the JiVE Server

No matter which JiVE configuration is used. Procedures for starting the server are the same. Note, a virtual machine for Java 1.1.x or higher must be installed in order to run the *JiVE_Server*.

In Windows, type the following at a command prompt from the installation directory:

```
start java JiVE/JiVE_Server
```

On a Unix machine, type the following from the installation directory

```
java JiVE/JiVE_Server &
```

If it is necessary to run the included web server, type the following as well, for windows:

```
Start java TinyHttpd <port_number>
```

Or for Unix:

```
java TinyHttpd <port_number> &
```

Where <port_number> is the port on which the web server will run (e.g 9999).

A.2.2 Adding New Users

To add a new user to the list of possible users in the system, open the Users.txt file (located in the JiVE subdirectory) in a text editor and add the person's user name and password separated by white space on a new line before the line with EOF.

A.3 Client

A.3.1 Running the Client Applet

To run the client load the index.html file (in the JiVE installation directory) in a Java 2 compatible appletviewer or web-browser. As of this writing the major web-browsers were just beginning to support the Java 2 platform, so we recommend the use of the Java 2 appletviewer.

If the server is using a standard web server, the user would type:

```
appletviewer http://<server_url>/.../<InstallationDirectory>/index.html
```

If the server is using the TinyHttpd web server, type:

```
appletviewer http://<computer_name>:<port_number>/index.html
```

If the server is going to run just on the local file system, change to the directory where the files are located (i.e. the JiVE installation directory) and type:

```
appletviewer index.html
```

A.4 Using JiVE

This section describes the use of JiVE for the design of a multi-agent system. It steps through the use of JiVE from the initialization of a project through the adding and editing of agents,

conversations, and conversation rules and finishing a project using the built-in design, analysis and code generation tools. The section finishes by describing JiVE's collaborative aspects.

A.4.1 Starting Off

The first step to working on a multi-agent system design in JiVE is starting the client and opening a project. Upon first opening the *JiVE_Client* applet a login window appears, as shown in Figure A-2. From this login window a user can create a new project or load an existing project. The login window has two modes, *New Project* and *Existing Project*, which can be toggled using the button in the lower right hand corner of the window (The figure shows the window in *New Project* mode).

A.4.1.1 New Project

To create a new project, type the name of the project and select the agent communication language (ACL) that the project will use. The name of the project can be any valid directory name for the server file system. The 'language' property of the project needs some more description.

The language property refers to the Agent Communication Language (ACL) that the multi-agent system will use. Three possible ACLs can be specified for a multi-agent system in JiVE. The default or 'Standard' language selection utilizes a subset of the JAFMAS *Message* class. This means that the designer must specify the 'Type' of each message and optionally may specify the 'Intent' of a message. In addition to the 'Standard' language option, there are pre-defined agent communication languages that are built into JiVE, 'KQML', and 'FIPA-ACL'. These choices preload a set of performatives into the set of available messages. These performatives conform to the standard set of performatives of KQML [Labrou and Finin 1998] and FIPA ACL [FIPA 1997]. For more information on ACLs in JiVE, see Galan [Galan 2000].

In addition to naming the project and selecting the desired language type, type the User Name and Password into the login window and then press 'Okay'. The *JiVE_Server* will validate the

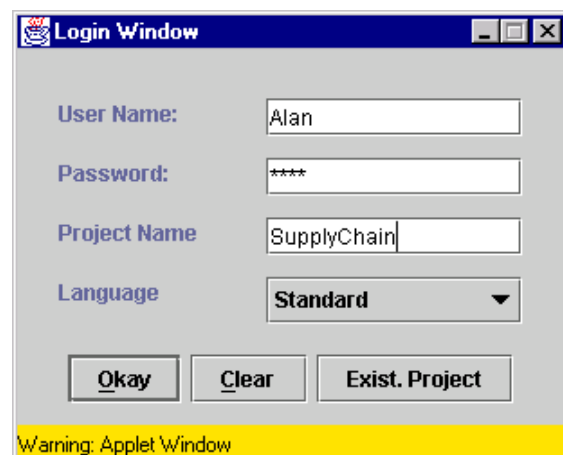


Figure A-2 - Login Window

login data and upon verification, the Multi-Agent System Design Window will open. This is where the properties and agents for the multi-agent system will be specified.

A.4.1.2 Existing Project

To open an existing project, select 'Exist. Project' from the login window, enter the User Name and Password, select the project to be opened from the pull down menu, and then press 'Okay'. The *JiVE_Server* will validate the login data and upon verification, the Multi-Agent System Design Window will open. This is where properties and agents for the multi-agent system will be specified and edited.

A.4.2 Designing the Agents

The Multi-Agent System Design Window is where the agents and the agent's properties are added, shown in Figure A-3.

A.4.2.1 Adding And Removing Properties

The first step is to add the agent's properties. These properties are the agents' 'Type', 'Attributes', and 'Subjects'.

The agent's 'Type' is a property that specifies the general agent type. For example, in a supply chain an agent might be a 'Customer' agent or a 'Plant' agent. The Multi-Agent System Design Window allows for the addition and removal of these properties.

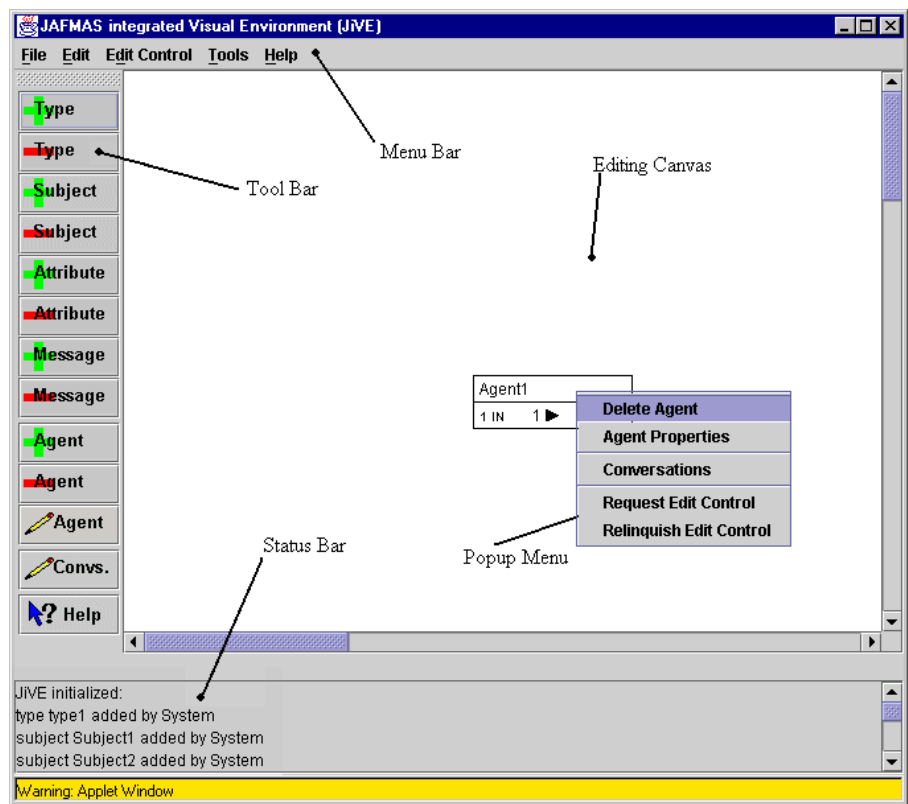


Figure A-3 - Multi-Agent System Design Window

To add an agent ‘Type’ property, push the

Type button on the toolbar or ‘Add Type’ on the ‘Edit’ menu. A dialog box will open allowing for the specification of the name of the Agent Type, as in Figure A-4. When ‘Okay’ is pressed, the request to add a ‘Type’ is sent to the JiVE Server. A message will be shown in the status bar indicating the success or failure of the operation.

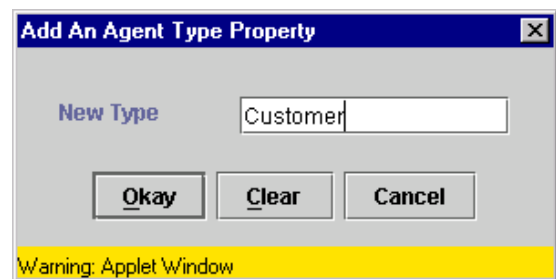


Figure A-4 - Add Property Window

To remove an agent ‘Type’ property, push the **Type** button on the toolbar or ‘Remove Type’ on the ‘Edit’ menu. A Remove Property window will appear where the ‘Type’ to remove is selected and the ‘Okay’ button is pressed. This submits the request to the JiVE Server. A message will appear in the status bar indicating the success or failure of the operation. The agent’s ‘Attributes’ are a set of one or more properties that classify what kinds of activities the agent can engage in or resources it can produce. For example, in a supply chain a ‘Plant’ agent may provide a ‘Paint’ service or a ‘Shipping’ service, or both.

The ‘Subjects’ that can be specified represent a subscription/publication channel, on which agents can send and receive multi-cast messages to other agent(s). The process to add and remove ‘Attributes and ‘Subjects’ are similar.

A.4.2.2 Adding and Removing Messages

When one of the pre-defined agent communication languages is in use (i.e. KQML or FIPA ACL) the adding and removing of messages is disabled. Otherwise, messages are added to the design in a similar manner as the other properties, however there is some additional information that can be specified for messages. To add a message to the design select **Message** from the toolbar or ‘Add Message’ from the ‘Edit’ menu. A window appears allowing the specification of the Message Type, which is the unique identifier for the message. This window also allows the specification of the message intent, which provides additional information as to what the message is meant for in the system but does not differentiate the message in JiVE. The messages are removed by selecting the **Message** button on the toolbar or ‘Remove Message’

from the ‘Edit’ menu. The window that comes up will allow for the selection of a message to be removed.

A.4.2.3 Adding and Editing Agents

Once properties and messages have been added, the agents themselves can be added to the design.

To add an agent, select **Agent** from the toolbar or ‘Add Agent’ from the ‘Edit’ menu, and click in the Editing Canvas.

This will bring up the ‘Agent Properties’ window, as shown in Figure A-5. In this window the particular properties that this agent is going to have are specified. First the

name of the agent is specified. Then, the type of this agent is specified by making a selection on the drop down box. One or more attributes that this agent can provide can then be selected.

Zero or more subjects can be selected to which this agent will listen. Finally, zero or more resources that are required by this agent can be selected, in the required resource list. Press ‘Okay’ to submit the request to the JiVE Server. A message with the results of the operation will appear in the status bar, and if the operation was successful then a graphical representation of an agent will appear in the Editing

Canvas. The agent will look similar to those shown in Figure A-6

To edit the properties of an agent already added to the design, click

 **Agent** from the toolbar or select ‘Edit Agent’ from the ‘Edit’ menu, and then select the agent to be edited; or right click on the Agent and select

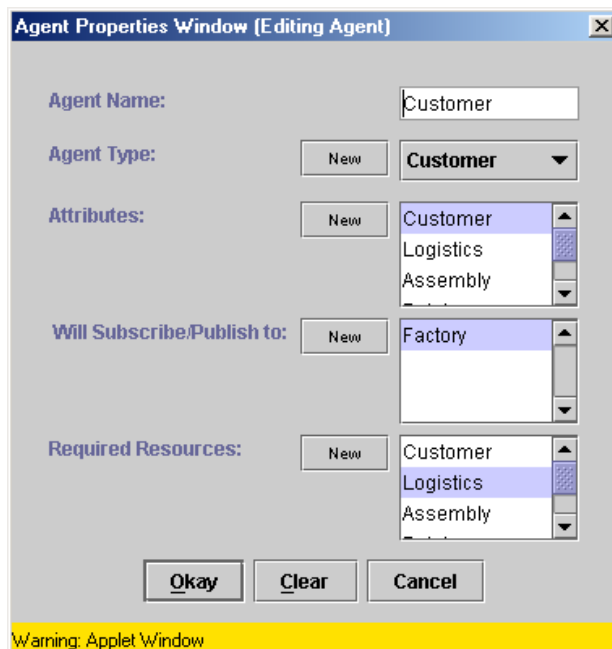


Figure A-5 - Agent Properties Window

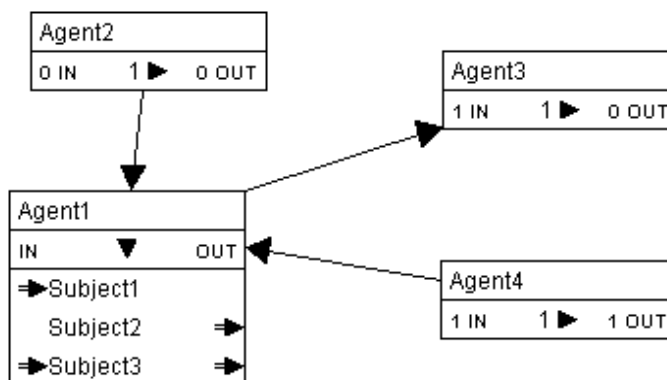


Figure A-6 - Agent Representation

‘Agent Properties’ from the popup menu. The Agent Properties window will open and the agent’s properties can be edited (Except for the agent’s name, which must stay the same). Continue to add agents to the design in this way.

A.4.2.4 Agent Representation

In JiVE, a rectangle on the Agent editing screen represents the agent. This rectangle lists the name of the agent, and an ‘expand’ icon, along with the number of intended subscription/publication channels. When expanded one can see the relationship between the agent and the various subscription/publication channels with which it communicates. For example, in Figure A-6 we see a depiction of an agent. This agent’s name is ‘Agent1’. The subscription/publication channels that it subscribes to and/or publishes to are listed below it on the expanded listing. We can see that ‘Agent1’ receives messages on the ‘Subject1’ subscription channel, sends messages on the ‘Subject2’ publication channel and both sends to and receives from the ‘Subject3’ subscription/publication channel. Also noted in this graphic are the directed communications that this agent has with other agents in the system. We see two arrows going out (to ‘Agent3’ and ‘Agent4’) and two coming in to the agent (from ‘Agent2’ and ‘Agent4’). With this information for each agent appearing on the Agent Editing screen, we can see the interactions between the agents, but not overwhelming details about the substance of those communications.

A.4.3 Designing the Conversations

For each agent added, the agent’s conversations must be specified. The conversations for a particular agent, including the states and conversation rules are added and edited from the Conversation Design Window, as shown in Figure A-7. To bring up the Conversation Design Window and edit an agent’s conversations, click the  **Convs.** button on the toolbar or on ‘Edit Conversation’ from the ‘Edit’ Menu.

The Conversation Design Window allows for the adding and editing of several entities. First, there are the conversations themselves, labeled rectangles, with a side bar on each side. There can be multiple conversations on a Conversation Design Window. These conversations represent the entire set of communications for the corresponding agent (i.e. the conversations in which the agent can engage.)

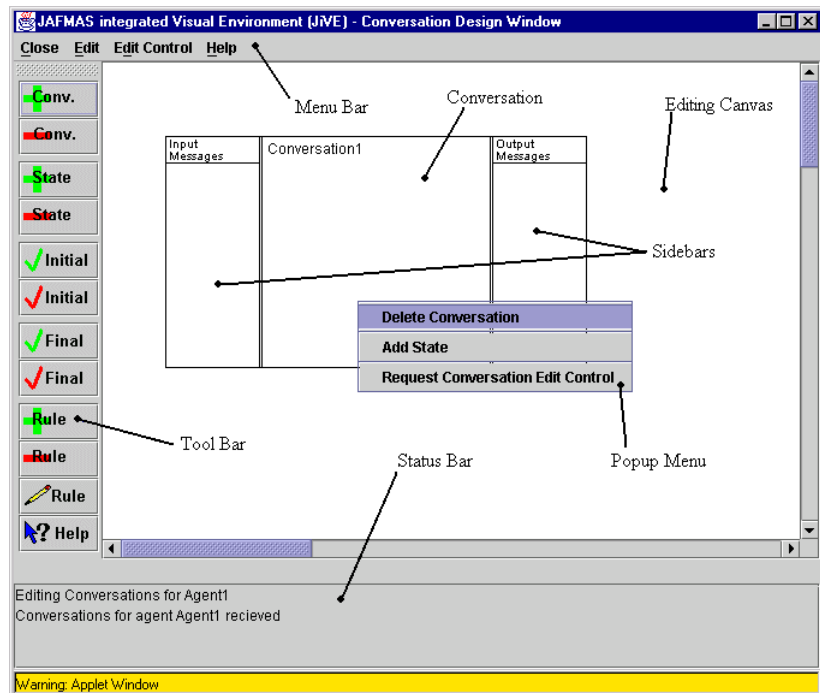


Figure A-7 - Conversation Design Window

The second entities in the Conversation Design Window are the states, labeled circles within a conversation rectangle (see Figure A-8.) The states represent the possible states of execution that a given conversation can be in at any one time. Each conversation will have multiple states. The final entities in the Conversation Design Panel are the conversation rules. A conversation rule, represented as a directed line, connects two states. The arrow points from the beginning state to the ending state of the conversation rule. The design of conversation rules is discussed in Section A.4.4.

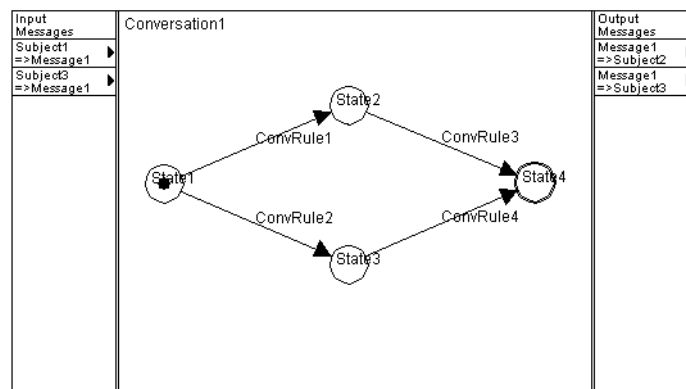


Figure A-8 – Conversation Representation

A.4.3.1 Adding and Editing Conversations

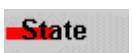
To add a conversation select from the Conversation Design Panel toolbar

 or 'Add Conversation' from the 'Edit' menu and then click and drag, drawing a rectangle on the Editing Canvas (drag from the top left to the bottom right of the rectangle.) A window will appear allowing the name of the conversation to be typed. Press 'Okay' to submit the request to the JiVE Server. The JiVE Server will return a message, displayed in the Status Bar, indicating the success or failure of the operation. If successful a labeled rectangle with a side bar on each side will appear on the Conversation Design Window's Editing Canvas.

To remove an agent's conversation from the Conversation Design Window, select  from the toolbar or 'Remove Conversation' from the 'Edit' menu, and click on the conversation to be deleted. (Note: Removing a conversation will also remove all states and conversation rules in that conversation). The conversations can be moved around the Conversation Design Window's Editing Canvas by just clicking and dragging on any empty area of the conversation rectangle.

A.4.3.2 Adding and Editing States

After adding a conversation, states can be added to the design. To add a state, select  from the toolbar or select 'Add State' from the 'Edit' menu and click inside one of the conversations. This will open a dialog box where the name of the state can be specified (the name of the state must be unique for that particular conversation). Boxes indicating whether or not the state should be an 'Initial' state or a 'Final' state can also be checked. Clicking on 'Okay' submits the request to the JiVE Server, which responds with a message indicating success or failure in the Status Bar. Upon success, a circle will appear in the conversation, with the name of the state.

To remove a state, select  from the toolbar or select 'Remove State' from the 'Edit' menu and then click on the state to remove. The status bar will show the success or failure of the operation and the state will disappear from the Editing Canvas. (Note: Any conversation rules connected to that state will also be deleted.) Additionally, states may be moved within their conversations by clicking and dragging the state.

A conversation must have an initial state and one or more final states. An initial state means that when the conversation starts it is in that state and is indicated by placing a token (a black

dot) in the state. A final state means that upon reaching that state the conversation terminates. Doubling the line that comprises the state indicates a final state. The initial and final status of each state can be toggled with the following procedure. For example, to make a state initial, select  from the toolbar or 'Mark State As Initial' from the 'Edit' menu and click on the state which will be marked as an initial state. The server will respond with a message indicating any changes made. This same procedure is applicable to 'Mark State As Final' -

, 'Unmark State As Initial' - , and 'Unmark State As Final' - .

After the various conversations and states have been added to the Conversation Design conversation rules can then be added to the conversation.

A.4.4 Designing the Conversation Rules

The conversation rules in a conversation indicate to the system the operations that must be done for an agent's conversation to transition from one state to the next. In JAFMAS, the order of execution of a conversation rule is as follows:

- A message is received that enables this conversation rule, meaning it has a specific message type and has been received from a specified agent or subscription/publication channel.
- The doBefore method is executed.
- The status of all methods specified for the suchThat property is verified as true.
- The doAfter method is executed.
- A specified message is sent with a specific message type to a specified recipient, either an agent or a subscription/publication channel.

Not every item in the conversation rule's order of execution needs to be filled in for any given conversation rule.

A.4.4.1 Adding and Editing Conversation Rules

To add a conversation rule in the Conversation Design Window, select the  button from the toolbar or 'Add Conversation Rule' from the 'Edit' menu, click on the starting state of the conversation rule, then click on the ending state of the conversation rule. This will bring up the Conversation Rule Editing Window, as shown in Figure A-9. In this window, the properties

desired for this conversation rule can be specified. First, fill in the name of the conversation rule, noting that the name must be unique for that conversation.

From the pull down menu, select the agent or subscription channel, from which a received message comes, and specify the received Message Type, in order to enable this conversation rule. If these properties are left out then the conversation rule will automatically transition whenever its start state is reached.

Specify the agent or publication channel, to which a sent message goes, and specify the Message Type for that message. If this property is left out then the conversation rule will terminate in the ending state without sending any messages.

For assistance in determining the relationship between agents and subscription/publication channels, the button labeled 'Subjects' can be pushed. This will open a window showing which agents subscribe/publish to which channels and to which channels each agent subscribes/publishes, shown in Figure A-10. This window will additionally denote any subscription/publication channels that have not been used yet.



Figure A-9 - Conversation Rule Editing Window

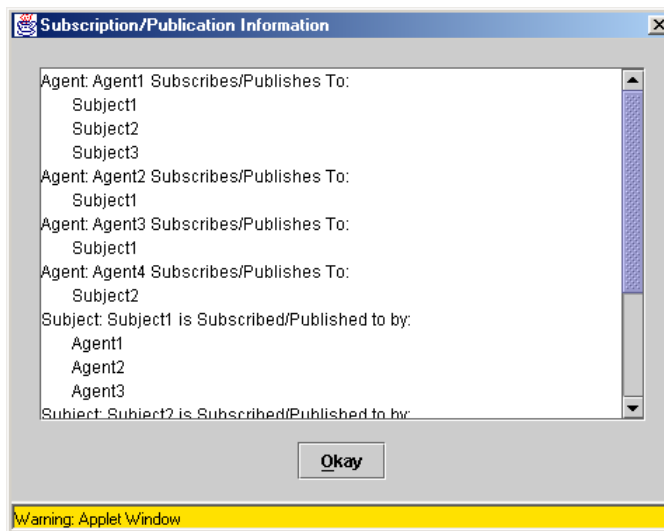


Figure A-10 – Subscription/Publication Information Window

The additional property that is set from this interface is the set of ‘suchThat’ methods. This is the set of methods, which must be true for the conversation rule to transition. (Note: Multiple methods are evaluated using a Boolean “And” operation). To specify the ‘suchThat’ methods, press the button labeled ‘suchThat’, and a window will open like the one shown in Figure A-11. For each method, type the name in the text field at the top and press ‘Add’. The name of the method is moved to the list on the bottom of the window. Each method name must be a valid Java method name. To remove a method name, select it in the list on the bottom and press ‘Remove’.

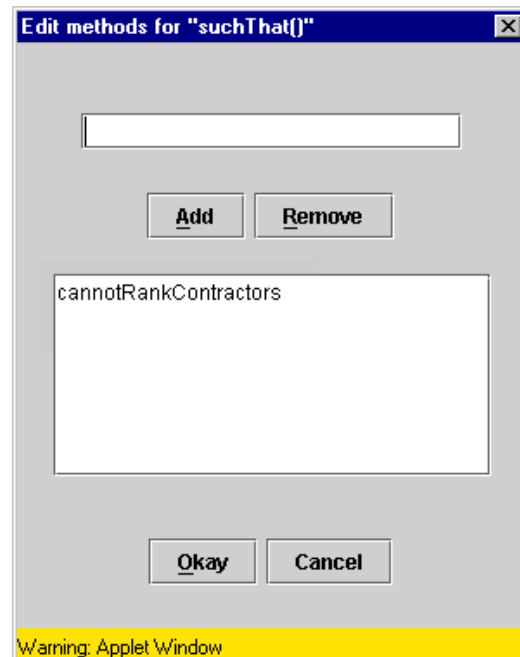


Figure A-11 - Such That Design

The method name is removed from the box. When finished specifying the methods for this conversation rule’s ‘suchThat’ property press ‘Okay’.

The doBefore and the doAfter methods are not specified via the JiVE GUI. It is instead left to the designer to fill in those two methods in the generated source code as desired (c.f. Section A.5.1).

Upon full specification of the conversation rule, the request can be submitted to the server by pressing ‘Okay’ on the Conversation Rule Design Window. The window will disappear and a message from the JiVE Server will appear in the status bar indicating the success or failure of adding the conversation rule. Upon success, a directed line appears in the Conversation Editing Window between the appropriate states. When the user desires more information on a conversation rule, the rule can be expanded to view all of its properties, by selecting  **Rule** from the toolbar or selecting ‘Edit Conversation Rule’ from the ‘Edit’ menu, and then by clicking on the conversation of interest. Other conversation rules can be added to the design in a similar manner. When the conversation’s design is complete, the conversation design window can be dismissed by selecting ‘Close Window’ on the ‘Close’ menu.

A.4.5 Analyzing Conversations and Generating Code

Once the entire design is completed, the system can be analyzed for coherency, and the JAFMAS source code can be generated. The analysis of the conversations is as easy as selecting ‘Analyze Conversations’ on the ‘Tools’ menu. During analysis, no changes to the design are permitted. In addition, the analysis can take some time depending on the intricacies of the design and the processing power of the machine in use. When the results are returned an alert will be provided concerning any problems in the design. These alerts will help track down any errors in the design. When the errors are fixed and the analysis is run again it returns a message stating that the conversations are ready to go, which means that the source code can now be generated.

The generation of the source code requires selecting ‘Compile Code’ from the ‘Tools’ menu, and if successful, the files are stored on the server, in the appropriate directory (c.f. Section A.1.2).

A.4.5.1 Tips for Designing a Multi-Agent System using the Analysis tool

There are several aspects of designing a multi-agent system using JiVE that will allow the analysis tool to work more effectively.

- Build the design incrementally. This will allow analysis checking as additional functionality is added to the design, and it will allow the analysis tool to assist in creating a coherent multi-agent system design.
- For conversations that may or may never be started according to the design, specify the Initial state as a Final state also. In the absence of this, the analysis will flag a deadlock condition.
- Avoid using a single subscription/publication message many different places in the design. This will tend to make the Petri net underlying the analysis grow very large. The result is an analysis that will take a long time to complete.

A.4.6 Collaboration

The collaborative features of the design environment are in many cases transparent to the user. That is, the user does not need to worry about edit control for the entities on which they are working.

The access control is applicable at several different levels, namely:

- Agent Edit Control
- Conversation Edit Control
- Conversation Rule Edit Control

That means that any one user can have a lock on any one of the entities listed, while other users can have the editing control for other entities. For example, ‘User1’ could have control of ‘Agent1’, while ‘User2’ could have control of ‘Conversation1’, which may be a conversation of ‘Agent1’. Of course, when any user in the system makes a change to any part of the design, this change is visible to every other user in the system.

The automated access control follows the following rules:

- As a user adds any controlled entity to the system, they are given access control automatically.
- If no other user has access control of an entity then any user who attempts an editing operation automatically receives access control to that entity.
- If a user has control of an entity then other users are prohibited from making changes. In this case, the users can use JiVE to request control from the user who has it. JiVE prompts the user with control with a dialog box. That user can relinquish control or ignore the request.
- When a user logs out of the system and/or the client loses the connection with the server, that user relinquishes Edit Control of all entities.

A.5 Multi-Agent System Deployment

This section outlines the method to create and deploy the multi-agent system once JiVE has been used to design, check, and generate code.

A.5.1 Adding Additional Functionality

After automatically generating the source code with JiVE, additional functionality can be added to the source code if desired. In some cases, there is some code that needs to be added in order to compile and deploy the multi-agent system.

To add additional functionality to the JiVE generated source code it is important to note that any editing that is done between the opening double brackets and closing double brackets will be deleted upon regeneration of the source code.

For example:

```
//{{ System defined fields: Do not edit
    Any additions in here will be deleted
//}}
    Any additions out here will remain
```

The only required additions are the `suchThat` methods, if they exist. If ‘`suchThat`’ methods are specified in the multi-agent system design, in any of the conversation rules, then the corresponding method definition will have to be added to the generated source code. In most cases, additional functionality will be added beyond that.

The `doBefore()` and `doAfter()` methods, which are included in the automatically generated source code, can be filled in if desired by the user. This would be the location to do any operation or call any methods that should be done before or after the operation of the conversation rule.

These methods can be found in the classes for each conversation rule.

A.5.2 Compiling the JAFMAS Source Code

To compile the JAFMAS Source Code place all of the JAFMAS files along with all of the JiVE generated files into the same directory. Execute the following:

```
javac *.java
```

Follow the directions in the JAFMAS System Administration and Users Manual [Chauhan 1997]. The only exception is when starting the agents themselves, as described next.

A.5.3 Starting the Agents

Agents generated by JiVE use a slightly different mechanism for launching than those using the original JAFMAS. This is because using JiVE generated agents relieves the user from having to use the ‘`CreateAgent`’ interfaces as specified in JAFMAS. In fact, JiVE allows the specification of the agents to create at design time. The recommended method for deploying the application is outlined here; specific applications may have different needs.

Each agent created with JiVE has a main method included such that it can be directly run the agent from the command line. An agent called 'Agent1' in the file called Agent1.java would be compiled as above and executed with the following command in Windows:

```
start java Agent1
```

Or in Unix:

```
java Agent1 &
```

Each agent in the JiVE generated system must be started using these commands in order to deploy the multi-agent system. It may be useful to create a batch file to start all agents in one command, this would of course incorporate system specific constructs such as paths and classpaths.

A.6 References

- [Brink 1996] R. S. Brink, "A Petri Net Design, Simulation, and Verification Tool." MS Thesis. Department of Computer Engineering, Rochester Institute of Technology, September, 1996. (Available at <http://www.csh.rit.edu/~rick/thesis/thesis.html>)
- [Chauhan 1997] D. Chauhan, "JAFMAS: A Java-Based Agent Framework for Multi-Agent Systems Development and Implementation." Masters Thesis. ECECS Department, University of Cincinnati, July 16, 1997. (Available at <http://www.ececs.uc.edu/~abaker/JAFMAS/index.html>)
- [FIPA 1997] FIPA, "Part 2: Agent Communication Language." *FIPA 97 Specification, Foundation for Intelligent Physical Agents*, Geneva, Switzerland, November 28, 1997. (Available at <http://drogo.cselt.stet.it/fipa/f7a12.zip>)
- [Galan 2000] A. Galan, "JiVE: JAFMAS integrated Visual Environment" Masters Thesis. ECECS Department, University of Cincinnati, 2000. (Available at <http://www.ececs.uc.edu/~agalan/JiVE/index.html>)
- [Labrou and Finin 1998] Y. Labrou and T. Finin, "Semantics for an agent communication language." *Intelligent Agents IV (LNAI Volume 1365)*, Eds. M. P. Singh, A. S. Rao, and M. J. Woolridge, Springer-Verlag: Berlin, Germany, 1998. 209-215.
- [Niemeyer and Peck 1997] P. Niemeyer and J. Peck. *Exploring Java, Second Edition*. O'Reilly & Associates, Inc. Sebastopol, CA, 1997.